

University of Edinburgh
College of Science and Engineering
School of Engineering and Electronics

Master of Engineering Project Report Part II

Design and Simulation of a Reduced State Microthreaded Processor Architecture

by

Oscar Almer

Supervisor: Björn Franke, School of Informatics

Edinburgh, 2007

MEng Project Mission Statement: Design and Simulation of a Reduced State Microthreaded Processor Architecture

Oscar Almer

18-11-2006

Student: Oscar Almer

Supervisor: Björn Franke

Subject area: Processor Design, Computer Architecture

Background

Processors are classically statically pipelined and executed instruction streams have inherent macro state. These properties have been somewhat eroded with the advance of processor design, but most processors still have structures based on these properties. Statically pipelined inherently means that each instruction must pass through all stages of the pipeline, which simplifies instruction issue. Execution stream state means that the processor must save and restore the macro state when switching between tasks or threads, but allows for shortcuts in instruction execution. If these properties could be reduced, it would open the way for inherently multi-threaded processing cores, at the potential cost of slower instruction issue and more complex execution units.

Aims

- To demonstrate the theoretical viability of a reduced state microthreaded processor core.
- To design a proof-of-concept reduced state microthreaded processor core.
- To successfully simulate such a core using suitable tools.
- To successfully generate a hardware description for building such a core.
- To generate a prototype compiler that will create viable code for the simulated core.

Scope for extra work

- Simulating the designed core in hardware (FPGA).
- Extend compiler to be able to port existing usable code to instruction set / processor.

Resources required

LISATEk tools

The supervisor and student are satisfied that this project is suitable for performance and assessment in accordance with the guidelines of the course documentation.

Signed

Student:

Supervisor:

Date:

Abstract

Due to advances in manufacturing technology, single thread processors are seen as less viable and efficient than multi-thread cores. The main problem is the reduced state accessible in a single clock cycle when faced with an increasing clock rate and shrinking architecture. This project proposes a novel multi-threaded reduced state processor design and investigates the viability of the proposed design. This design couples the registers with the instructions being executed, and greatly reduces the amount of global state kept in the processor. The reduction in global state allows for the processor to execute many instructions from different threads at once, thus enabling massive multithreading. This kind of processor is then shown to be viable in construction, as a prototype is constructed and is demonstrated to be working in simulations as well as in a hardware fabric. The associated compiler for the processor prototype is also constructed and shown to generate executable code for the processor. The combined system is suited for applications which benefit from parallel processing and enables multithreaded performance gains to such applications while having a relatively simple hardware construction.

Statement of Originality

I declare that this thesis is my original work except where stated.

.....
Oscar Almer

Contents

Mission Statement	ii
Abstract	iii
Statement of Originality	iii
Contents	iv
List of Symbols and Abbreviations	vi
1 Introduction	1
1.1 Concept and Goals	1
1.1.1 Project Goals	1
1.1.2 Processor Design Concept	2
Reduced State	2
Microthreading	2
Instruction Blocks	3
1.1.3 Compiler Design Concept	3
1.2 Related Processor Designs	3
1.2.1 Superscalar Processors	3
1.2.2 SMT processors	4
1.2.3 VLIW processors	5
1.2.4 Multi-core Interconnected Processors	6
1.2.5 Vector Architectures	6
1.2.6 Tiled Architectures	7
1.2.7 Dataflow Architectures	8
1.2.8 Microthreaded Architectures	8
1.3 Architecture Comparison to MTRS	9
1.3.1 Microthreading	9
1.3.2 Reduced State	10
1.3.3 Compiler	10
1.4 Plan	11
1.4.1 Prototype processor	11
1.4.2 Prototype compiler	11
1.5 Summary	12
2 Processor	13
2.1 Original Design	13
2.1.1 Arithmetic operations	14
2.1.2 Multithreading	15
2.1.3 Register Size	16
2.1.4 Instruction size	17
2.1.5 Memory Manager	18
2.1.6 Initial Design Decision Summary	18
2.2 Modified Design	19
2.2.1 Instruction and Word Sizes	19
2.2.2 Memory Interface	22
2.2.3 Processor size	23
2.2.4 Summary	24
2.3 Final Implementation	24
2.3.1 Load Functional Unit	26
2.3.2 Store Functional Unit	27
2.3.3 Compare Functional Unit	28
2.3.4 Add Functional Unit	29
2.3.5 Nop Functional Unit	30
2.3.6 Spawn Functional Unit	31
2.3.7 Instruction Decode	33
2.3.8 Interconnect Network	34

2.3.9	Serial Output	34
2.3.10	Summary	35
2.4	Testing	35
2.4.1	Verilog Unit Tests	35
2.4.2	Processor Simulations	36
2.4.3	Full-system Software Simulations	36
2.5	Summary	37
3	Software Tools	38
3.1	Introduction	38
3.2	Assembler	38
3.2.1	Motivation	40
3.2.2	Language Choices	40
3.2.3	Implementation	41
Linker Implementation	41	
Assembler Implementation	42	
3.2.4	Testing	43
3.2.5	Summary	44
3.3	Program Loader	44
3.3.1	Hardware	44
3.3.2	Software	45
3.3.3	Summary	46
3.4	Compiler	46
3.4.1	Motivation and Approach	47
3.4.2	Compiler structure	47
Translating to Threaded Code	48	
Compiling to MIPS	50	
Offset and Register Mapping	51	
Function Call Convention Replacement	53	
Instruction Replacement	55	
Subinstruction Mapping	56	
3.4.3	Testing	58
3.4.4	Code Quality	59
3.4.5	Summary	60
4	Conclusion	62
4.1	Processor	63
4.2	Compiler and Assembler	64
4.3	System	65
	Acknowledgments	66
	References	67
	A.1 Appendix: Instruction Reference	68
A.1.1	Development	68
A.1.2	General instruction format	68
A.1.3	Reference Assembler	69
A.1.3.1	Non-Instructional Operatives	70
.CALL	70	
.SET	70	
.VAR	70	
A.1.3.2	Sub-Instructions	71
Nop	71	
Store	71	
Conditional	71	
Load	72	
Add	72	
Spawn and Jump	72	
A.1.3.3	Example	73

List of Symbols and Abbreviations

Abbreviation	Description	Definition
DSP	Digital Signal Processing	page 5
FPGA	Field Programmable Gate Array	page 1
IPC	Instructions Per Clock	page 3
ISA	Instruction Set Architecture	page 15
MMU	Memory Management Unit	page 1
MTRS	Micro-Threaded Reduced State	page 2
SMT	Simultaneous Multi-Threading	page 4
VLIW	Very Long Instruction Word	page 5

1 Introduction

1.1 Concept and Goals

The project is to design and simulate a processor core in order to show that the underlying idea for the processor is viable. The processor concept is in itself innovative, and thus the project is primarily aimed at building a proof of concept general purpose processor using the ideas outlined herein. The theoretical foundation for the processor and the simulation methods and tools were the subject of the first part of this Project Report [1].

1.1.1 Project Goals

The goals for this project are to create, simulate, and demonstrate the workings of a prototype reduced state microthreaded processor core, and to further create a prototype compiler capable of generating efficient code for such a core. The core will not be implemented as a single design in silicon as part of the project, instead the goal is to simulate it as accurately as necessary and then use *Field-programmable Gate Array* (FPGA) technology to realise a working implementation. FPGA technology allows for the creation of arbitrary digital logic blocks, and can be used to simulate digital electronics designs at clock speeds an order of magnitude lower than optimised silicon circuitry, but without the costs of creating silicon circuitry for the application. FPGA technology is also reprogrammable, meaning that the design can be evaluated and modified if needed with very small overhead.

It is notable that the design of a suitable *Memory Management Unit* (MMU) for this processor core is considered to be outside the scope of the project, and as such, the finalised design cannot be easily adopted for any purpose other than proving the design of the processor itself. The concepts, once proven by the prototype implementation, can be carried as necessary.

The compiler for the prototype implementation is likewise not intended to be anything but a proof-of-concept, and will thusly lack many advanced features such as automatic code optimisation. It is intended only to be able to compile basic programs to execute on the processor, though it should be able to perform this task automatically. It is expected that the compiler is only able to translate from a moderately low level program description to executable code, as long as this can be further extended to higher level languages at a later time.

As well as compiling code, there is a need for a software toolchain to assemble, link and lastly load the code into the FPGA unit the processor is implemented in. These functions are all separate but are still part of the project as the compiler and the processor design would be unusable without these interconnecting steps. These functions also have to be implemented together with the compiler as part of the software side of the project.

1.1.2 Processor Design Concept

The design concept is for a *Micro-Threaded Reduced State* processor core (MTRS). Intended as a processor core, it should have the basic properties of such a device, namely being able to read from and write to a memory of undefined size, perform some arithmetic functions on data values, follow a thread of execution loaded from the same memory, and also change thread of execution depending on previous instructions. This effectively amounts to Turing completeness of the design, which is obligatory if the design is to be considered a processor at all. Further, as a general-purpose processor, the design should also be a von Neumann architecture. This architectural model uses the same memory for both instructions and data, and as already outlined, this is the case for MTRS.

Reduced State

Reduced state refers to the internal state of the processor core, and that it should not store excessive state information relating to the instructions that are executing through it. Storing such information is expensive in hardware, since it uses particular hardware structures to retain information, which in turn increase the hardware complexity and thus drives up costs. It is also expensive in time when switching tasks and threads in the processor as such state will either have to be stored for each task or created anew for each task. Process specific register data is usually swapped out on task switches, costing time and memory bandwidth, while branch predictor data is commonly discarded just to be recreated when the task is switched in again. Most common processors use both these techniques, depending on the exact use and importance of the state information, and thus incur heavy penalties on task switching. With a reduced-state processor, these costs are minimized, leading to improved task switching performance. The key goal is that the working state in the processor should not be coupled heavily with the currently executing instruction stream.

Microthreading

Microthreading refers to the use of short-lived, easily created and destroyed threads of execution. These threads should be cheap enough that turning an iterator loop into parallel microthreads should be an easy and natural development. The processor core should support many simultaneous microthreads, allowing the iterator loop to execute in significantly shorter time. This implies that single-threaded program performance may suffer in this design, but this is acceptable in order to improve total thread throughput. This synergises with the reduced state property, as task switches to and from these threads are fast and cheap.

Instruction Blocks

MTRS also makes use of instruction blocks consisting of subinstructions. Subinstructions are the equivalent of functional units in common processors, and an instruction block is roughly equivalent to a full instruction in a conventional superscalar processor. This allows instruction blocks to schedule to functional units that are free, allowing multiple functional units and multiple microthreads taking advantage of the same type of functional unit at any one point. Termed dynamic resource binding, this property increases throughput of microthreads, and decouples instructions from a fixed pipeline. This further cooperates with the reduced state and microthreading properties, allowing distinct microthreads to intermingle and allocate to different equivalent functional units, thus increasing resource utilisation.

The concept is to issue many instruction blocks in the same clock cycle into the instruction path from different, possibly unrelated, instruction streams or threads. This leads to possibly reduced single thread performance, but good multithread and transaction performance. The resulting processor core is therefore geared towards massively parallel systems, such as database services or scientific computing. Certain computer graphics rendering modes are also expected to benefit from this concept.

1.1.3 Compiler Design Concept

1.2 Related Processor Designs

The processor design has some shared structures with VLIW style designs, but differs substantially in other aspects due to the outlined concepts. VLIW uses static binding of instructions to functional units, while MTRS uses dynamic binding; VLIW execute blocks of independent instructions while MTRS issue blocks of instructions that may or may not be dependant. MTRS is also related to SMT ideas, but again the practical differences are significant. Due to the largely parallel nature of the design concept, it might be apt to compare the design to a supercomputer; this however breaks down due to the highly specialised nature of all supercomputer designs, and no single relevant comparison can be made. Various other processor execution models are also more or less relevant to the MTRS concept and are included for completeness.

1.2.1 Superscalar Processors

Originally processors processed one instruction at a time, and this formed the basis of the performance model to which all processors still measure; how many instructions can be retired in a single clock cycle, or *Instructions per Clock* (IPC). The pipelined processor was an improvement on this, but was also originally limited to one instruction completed per clock. This was further im-

proved by multiple-issue enhancements, branch predictors and deep pipelining. It is still however the case that each instruction goes serially through each pipeline stage, even if there are multiple instructions passing through the pipeline at once [2]. In essence, pipelining increases instruction throughput but declares that all instructions takes as long to execute; it may increase the execution time of any single instruction but also increments instruction throughput leading to an overall faster processor. To take full advantage of a pipeline there is a need for a complex branch predictor, register renaming hardware, and other non-trivial additions. These improvements all increase the IPC of the resultant processor, and developments in this area are what have driven processor technology to its current point. This type of processor is referred to as a superscalar processor, as it operates on more than one instruction and data item at once.

The MTRS design concept differs significantly from the superscalar model, in that it has no fixed pipeline, and is severely multithreaded, in addition to operating on many instructions and data items at once. The idea of increasing throughput by overlaying many instructions is however the same. The crucial difference is that in MTRS the instructions are not necessarily related in any fashion, and that MTRS emphasises multi-thread performance over single-thread performance, which is the goal of superscalar processors.

1.2.2 SMT processors

Symmetrical Multithreading (SMT) processors take the concept of a pipelined processor and add multithreading concepts on top. Observations showed that the pipeline was for various reasons, mostly related to instruction scheduling and imperfect prediction, not often filled to capacity, and that the spare slots in the instruction pipeline were wasted, leading to low utilisation of functional units and thus low performance compared to the theoretically possible. It was realised that these spare instruction slots could be used for a second thread of execution. It was shown that this could be done [3] with minimal single-thread performance degradation, and this technique was for a few years widely used as it could be added to an existing microarchitecture at a very small cost in hardware complexity and money. The concept was named 'Hyperthreading' by Intel, and was integrated into their line of general purpose processors to improve multithreaded performance. The brand name was heavily marketed and enjoyed some popularity. It did not fall out of favour until the release of true dual-core processors, but even so some of the dual-core processors subsequently released still had SMT / Hyperthreading capability.

In effect, this development presents a single physical processor as two logical processors to the software, but each possible thread of execution competes for space in the same pipeline, thus utilizing the available pipeline resources better and increasing IPC. SMT enabled processors work on the granularity of entire processes and/or classical heavy threads when addressing the processor utilisation problem. The disadvantage is that there has to be more than one thread of execution,

otherwise the performance is equal to or slightly worse than the same processor without SMT enhancements. The emphasis in this development was better use of available processor resources through multithreading at low cost in complexity.

SMT Processors are related to the MTRS amalgam in the shared goal of trading unused parts of the processor to execute another thread, or several threads; it contrasts in the number of threads thus enabled. SMT is designed as an addition to pipelined processors with a fixed microarchitecture, while MTRS is designed as a stand alone processor. MTRS can be considered as an enhancement on the SMT concept, if development were allowed to go forward without regard to single-threaded performance.

1.2.3 VLIW processors

Very Long Instruction Word (VLIW) processors is a development [4] in which the compiler is coupled tighter into the architecture. Instruction scheduling is done entirely in the compiler, and the processor is just an execution engine. This replaces the expensive and complex scheduling hardware necessary for an out of order deep pipeline processor with a simpler and therefore cheaper issue unit. This type of architecture schedules several independent instructions at once for execution, but they all come from the same thread. The compiler is responsible for choosing instructions that can be executed in parallel without dependencies. The compiler cannot make any inference of what other threads are doing at runtime, as it has no control over those threads, and thus the instruction schedule is by necessity single threaded. This type of processor tends by necessity to have functional units exposed directly to the compiler in a greater fashion than superscalar processors. The compiler schedules instructions directly to functional units in a clock cycle, and has full control over the instruction flow in the processor.

VLIW processors tend to keep as much state as Superscalar processors, and are therefore particularly inefficient when considering task or thread switches due to the amount of state that needs to be swapped out and in again. VLIW is however a popular design choice for *Digital Signal Processing* (DSP) processors, as their real time needs benefit from the simple resource binding and never need to context switch. There are several lines of VLIW DSP processors commercially available, with brand names such as TriMedia. While not intended for general purpose these processors are fairly successful in the embedded market.

In contrast, a MTRS compiler schedules instructions to a series of functional units, but does not assign instructions directly to functional units. Instead, it sets the order of functional units the instruction block will follow through the processor, and leaves the exact path and decisions to the MTRS processor itself. This dynamic resource binding is half way between the superscalar and VLIW extremes, in that it is neither fully determined nor undetermined at compile time which exact functional units will be visited. This allows any instruction block to mix with other unrelated

instruction blocks without conflicts, thus allowing the microthreaded property of the processor.

1.2.4 Multi-core Interconnected Processors

Multi-core processor systems are generally systems that have more than one distinct processor unit, and that have the processor units cooperate either through hardware, software or a combination of the two. In general, these systems are massively multithreaded, but the threads are classically heavy and have significant startup and termination costs. At a microarchitecture level these systems are not necessarily different from a superscalar processor, and thus share the microarchitectural properties, be they Superscalar or VLIW. Some of these interconnected processors are intimately coupled in hardware, but are still considered two separate processors architecturally as threads can schedule to them independently and without affecting performance of threads scheduled to other processors. This is in contrast to SMT which are not considered two architectural processors as the threads go through the same pipeline and have an impact on each other. Processors arranged in a multi-core system may share memory and caches, but they have distinct and separate pipelines. A popular example of this type of processor is the Sun Niagara line of processors [5]. A single physical Niagara chip has 8 simple superscalar processors on it, which share caches and off-chip connects. Each superscalar is capable of handling up to four threads simultaneously, hiding memory access latencies by executing other threads. This arrangement allows 32 independent threads to execute at once. A later refinement of the concept, dubbed the T2, introduced 8-way capabilities for each of the 8 pipelines, allowing a maximum of 64 threads at once on a single chip. As can be seen, this shares the goals of MTRS of thread throughput as opposed to single-thread performance, but it is still relying on classical heavy threads.

Multi-core systems, in particular the closely coupled designs, are different from MTRS on one critical point. MTRS uses a single pipeline for all microthreads, whereas these systems use several independent pipelines. The difference means that the hardware resources are more finely allocated to threads in MTRS, as all threads have access to all functional units. This can lead to better resource allocation in MTRS, allowing either transistor counts to decrease to save power at the same performance level or allow thread count to increase with resultant increased throughput.

1.2.5 Vector Architectures

Vector processors are substantially different from scalar processors in that while scalar processors act on a single data value at a time, vector processors apply the same transformation to a number of data elements at the same time. This type of processor used to be common for scientific applications as it is optimal for performing operations on large datasets, and is able to do so more efficiently than a scalar or superscalar processor. However, for control flow and related operations the vector processor offers no advantage, and it is therefore less suited as a general purpose

processor. Vector architectures have been the purview of supercomputers, where control flow mattered less than pure computation speed. The supercomputer company Cray in particular build and successfully marketed Vector architecture computers, including the well known CRAY-1 [6]. The market for these architectures was however shrunk due to the improvements subsequently made on superscalar processors and the use of multi-core interconnected systems of cheaper Superscalar processors. General-purpose processors have also seen the introduction of vector operations into their instruction sets, allowing them to benefit from limited vector optimisations while retaining their original instruction set, which has improved their performance in these types of operations.

Vector architectures can be emulated using multithreaded or microthreaded architectures. Individual vector operations are farmed out to many scalar processors which complete the operations in parallel. This type of computing has the advantage of granularity of operations and portability of code, in that the code is no longer limited to certain vector lengths and is thus no longer tied to a certain machine vector length. Because of this modern multithreaded architectures are more desirable than vector architectures.

MTRS is as outlined a microthreaded scalar processor, and thus is not directly related to vector processors. It is however possible to implement MTRS with vector operations, either via emulating the vector operations as microthreads or by use of vector functional units. Implementing it as microthreads is the preferred option as this requires less specialization of the processor and is expected to interact better with general control flow.

1.2.6 Tiled Architectures

Tiled architectures consist of a number of interconnected processing units sharing the same physical substrate, laid out in a tile of sorts, which gives rise to the name. These processing units are in general connected to a few adjacent processing units for passing data. This type of processor is currently not widely used in any field, but could become useful. The main point is that the processing units can communicate with the neighbouring processing units very quickly, allowing extremely efficient communication between both threads and processes which are running concurrently on adjacent tiles. Tiled processors are thus particularly suited for heavily multithreaded programs, provided that the program is split over tiles in an appropriate way. Allocating threads to tiles is however a non-trivial problem, particularly as it has to be done in a dynamic environment as threads, and thus tile allocations, are spawned and retired.

Tiled architectures are sold commercially by at least one company, Tilera, which is currently marketing its TILE64 processor. The technology is a commercialised version of the RAW project at MIT [7], which in turn is a development of an earlier MIT project. It is as of yet uncertain how this type of architecture will fare in a commercial setting.

These architectures are as stated rare, and sit conceptually closest to extended vector architec-

tures. Their relation to MTRS is faint as these architectures tend to have a lot of state internally in the processing units, and are generally running specially created single threaded programs. They are relevant as they utilise a programmable processing unit interconnect much like MTRS, though it is less general and therefore less complex in those architectures.

1.2.7 Dataflow Architectures

Dataflow architecture is a distinctly different idea to classical control flow architecture. Instead of explicit control flow and an instruction stream, Dataflow architectures let all instructions be eligible for execution when their arguments are ready. Each instruction is tagged with which arguments it requires and what data it produces, and is scheduled for execution as soon as all dependencies are fulfilled. The order that instructions execute in may be different from the program order specified in the source code, but the dependence graph of the program remains intact, thus ensuring correct execution. This paradigm was never used for a commercial computer system; there were issues of scale and cost that prevented a successful large scale implementation. There were research machines built to this concept [8], but due to problems with parallelising wide enough while still retaining performance they were never commercialised.

The dataflow concept is rather similar to what happens on a limited scale in a modern out of order superscalar processor, apart from the issue of scale. Dataflow architectures were intended to be much larger than the limited units utilised here. It is because of this scaling that it is possible and efficient to use the concept for mainstream processor architectures. The underlying idea is the same as full dataflow architectures; let an instruction on to execute only once its dependencies are complete, and utilise this to schedule instructions when they are ready to execute rather than in program order.

Dataflow is not directly related to the MTRS concept, but it is included as it makes multithreading trivial; a program can be executed in many parallel stages as the dependence graph is followed rather than the program order. A dataflow program can theoretically take advantage of as much parallelism as offered by the actual hardware, and do so in a fashion that is of finer granularity than classical threads or even microthreads.

1.2.8 Microthreaded Architectures

Microthreaded architectures are the subject of ongoing research, and have as yet not reached production stage though some experimental implementations exist. The main idea of microthreaded architectures is to let the compiler break legacy linear code into microthreads, which can be dynamically scheduled into one or several pipelines at execution time [9]. The benefit is that the compiler can extract parallelism from the code, as is done in VLIW compilers, but the code is dynamically scheduled at runtime, thus eliminating the issues with the static scheduling in VLIW.

A single classical thread may have many microthreads, which will interact in the hardware scheduler to provide an efficient schedule for the thread based on dynamic factors and the availability of data.

Other than breaking the code into fragments that can be independently scheduled, no effort is made towards changing architecture or reducing the processor state, as is done in MTRS. The result is that the architecture is stateful, just as a superscalar, and thus suffers from the same problems on thread switches. Microthread switches are handled easily, but use of classical threads in this model will lead to the same problems with context switches as in superscalar processors.

The MTRS concept is related closely to the Microthread concept, but the addition of the reduced state property changes makes the two different. The reduced state property makes context switches easier in MTRS, as there is little to no state to switch out. The compiler for MTRS has the same job as the compiler in microthreaded architectures, that is to split linear code into microthreads; the difference is that the hardware in MTRS allows microthreads from different threads and even different programs to schedule together. This leads to improved system performance in MTRS as opposed to improved single-thread performance for microthreading.

1.3 Architecture Comparison to MTRS

MTRS is related to many existing architectures, in particular it has been influenced by VLIW and SMT concepts as well as Microthreading. Further, MTRS is an amalgam of these separate concepts, taken from different architectural families, which have all influenced each other and turned into the solidified MTRS concept. The relation between MTRS and these separate concepts is outlined in this section.

1.3.1 Microthreading

MTRS is related to SMT, VLIW and Microthreaded architectures, which all have parallelism as one of their design goals. Microthreading architectures uses explicit multithreaded parallel execution to improve performance, and is very closely related to MTRS. SMT modifies an existing architecture to appear as parallel processors through inserting unrelated instructions in the pipeline, thus providing facility for concurrent threads. VLIW instructions are assigned to execution units by the compiler, which can parallelise loops very efficiently. MTRS uses all these ideas to reach a massively parallel goal; firstly by using a compiler to schedule code in microthreads, secondly by having a pipeline that can schedule several microthreads at once in a way that looks like SMT, and thirdly by using explicit functional unit addressing instead of a static pipeline like a VLIW. Thus the ideas behind microthreading, SMT and VLIW have been integrated into the MTRS system, and as can be seen they complement each other to produce a natively parallel architecture.

Parallelism is also done by a host of other architectures, such as Vector and other SIMD systems, but these are aimed at parallelising single threads and single instructions to work on more data rather than parallelising multiple threads together. Tiled architectures in contrast parallelise threads very well, but have issues with the mapping of threads to processors, and is comprised of many single-threaded cores. Further, multi-core architectures are optimised for many threads belonging to the same work set, and are superficially similar to MTRS; but they are significantly more coarse-grained in their thread allocation and execution.

1.3.2 Reduced State

The goal of reduced state is to reduce the amount of program-specific state that is stored in the processor, so as to facilitate context and task switches, something that is vitally important for microthreading. It is also crucial as manufacturing technologies go towards comparatively increasing wire latency, an intrinsic effect of shrinking transistor sizes. Even so, very few processor architectures have attempted the goal of reducing program state. Superscalar processors have for example been increasing their internal state for a long time, and VLIW processors, with the prime example of the Itanium, have a very large working state that needs to be replaced on context switches. In contrast, MTRS does not necessarily have a register file and very little state is preserved in the pipeline, but have the state associated with threads and any other useful state stored in the memory controller.

In this regard, the least related processor architecture is the dataflow model, whose basic idea is to hold all program state in the processor at once. This is in some ways the complete opposite of reduced state. Unfortunately no architecture can be listed as a good example of the reduced state property, as all currently existing architectures rely on internal program state to be efficient.

1.3.3 Compiler

The MTRS compiler is responsible for breaking linear programs into independent microthreads, and to ensure that correct execution is still accomplished. This is clearly related to the VLIW compiler ideas, which break programs into instruction sets that can be scheduled simultaneously. This is orthogonal to microthreading, which schedules instruction traces that are sequential and lets the hardware sort out the parallel scheduling; even though the methods are orthogonal, the compilers have a similar job and can use similar techniques to accomplish the goal. This type of compiler is needed for all microthreaded architectures.

Parallelising compilers are in general difficult to produce and most importantly it is hard to locate parallelism in linear code, but this is a problem that matters less for MTRS than other architectures as the emphasis is on system throughput rather than single thread throughput, placing slightly less demand on the MTRS compiler than a VLIW compiler. The compiler is thus an

important part of the architecture itself, to a higher degree than the compiler is in a superscalar architecture, and the compiler has to be able to generate efficient MTRS binary code.

1.4 Plan

The plan for the project was to first create a prototype MTRS hardware implementation, realised in a FPGA structure, then to create a compiler able to compile code for the hardware prototype. As can be expected, the plan did not hold; in particular, while the overall time limits assigned to the various tasks were generally held, the nature of the work resulted in several occasions where the hardware design had to be altered so that the compiler could be built more efficiently. The most important of these points are outlined in sections 2.2 and A.1.1.

1.4.1 Prototype processor

The plan to create the hardware implementation concentrated on one part of the hardware at a time, and built up the completed processor functional unit by functional unit. Testing was applied on a per-module basis; each module was verified to be working before work moved on. Time for each module varied on the complexity of the module; some simple modules were completed in a day or two whereas more complicated functional units took up to a week to complete. The spawn units in particular were the ones that took longest to complete, as can be inferred by their many interlinked inputs and outputs. In all, the hardware creation part of the project was assigned about 5 weeks worth of work, and this schedule was largely adhered to. The resultant prototype processor fulfils the goals set out in the Mission Statement, and further also fulfils one of the goals for extra work, namely a working implementation in a FPGA structure.

1.4.2 Prototype compiler

The compiler was scheduled to be worked on after the prototype processor, but as has already been stated the work ended up overlapping due to the requirements. The compiler is built in sections, starting with the lowest level (assembler) and working upwards, again so that each section could be tested before building the next step. The Compiler received less working time than the processor, and is therefore the less completed part of the project, but even so it is complete enough to be a proof of concept. The time assigned to the compiler was about 4 weeks, but this in the end became somewhat shortened, leading to the possibility of spending more time on improving the compiler. Nevertheless, the goals for the compiler as outlined in the project Mission Statement were met and the compiler is able to generate viable code for the MTRS processor.

1.5 Summary

The concept of reduced state microthreaded processors rely primarily on the decoupling of a processor from a process. The properties of the architecture have been chosen so that several processes and a large number of microthreads are allowed simultaneously in a single processor. It further relies on throughput versus single-threaded performance to speed up computing. The underlying assumption is that there are enough threads available in the computing system to load the processor to an appropriate degree.

This is different from currently common architectures that emphasise heavy-handed threading or even single threaded performance. The architecture can however be viewed as a VLIW processor that has had an advanced version of SMT added to it, and in this context it can be related to superscalar and other designs.

The project goals are to design and build a MTRS prototype implementation, and to show it working; in addition to also create a proof of concept compiler for the implementation. More effort will go into the processor design than the compiler design, as it is expected that the compiler can to some extent consist of previously available compilers.

The project was largely completed within the time limit, but more time could have been spent on various aspects, in particular the compiler, both because of time constraints and project focus. Even so, the compiler has all necessary features, the prototype MTRS implementation is functional and can execute code assembled using the created software toolchain.

2 Processor

The MTRS concept started as an extension to the VLIW architecture ideas, but was extended from there to be different enough to be labelled its own architecture. The important properties of the architecture is that it is massively multithreaded, and that the threads it supports are not necessarily related to the same program. The design concept is easy to understand, but realising this concept is harder. In effect, what is needed is a fabric of functional units interconnected so that they can pass instructions between each other in any order. By making instructions not visit each functional unit in turn, time and energy is saved. At the same time, multithreading is accomplished through instructions being independent. This has benefits for logical chip layout as clocks scale up and transistor sizes shrink. The benefits in this area are a direct consequence of the reduced state property. A problem that might remain to be overcome in this design is the clock signal distribution and the capacitive load on the clock signals, which is expected to be high as all functional units and the interconnects are synchronous to the same clock.

This chapter outlines the design and build process of the processor prototype in detail, and outlines how one might build a processor that embodies the design concept outlined. The theorised and real capabilities of the prototype is discussed and contrasted, and the internal workings of the resultant processor is examined. This is all to show that a processor following the MTRS design concept can be built and operate in accordance of the concept. The most important part of the MTRS concept is the interconnect between functional units. The design prototype presented herein will be using an reduced omega network, as discussed in the Phase 1 report [1].

Conventional architectures use a register file to store variables, and this has been a staple of processor design for many years. Due to the nature of the MTRS processor, it does not use a register file, but nevertheless has to store data values for performing operations on them. In MTRS this need is solved by storing the data in fields alongside the instruction itself, and the data is thus shipped around the processor together with the instruction. Despite not being stored in a register file, these data fields are still referred to as registers, as they fill the exact same function as registers in a register file in a conventional processor.

2.1 Original Design

The original design is centred around the omega network, and the design parameters and other data for the detailed implementation is taken from the size and properties of the omega network. Some decisions are orthogonal to the omega network concept, such as the instruction width and memory interface details. Note as before that the memory controller proper is outside the scope of the project, and as such is not covered by this section, but that load and store operations are a part of the instruction set and so are covered. These sections outlines the original design decisions for

the prototype processor, its subunits, and some indications of how these are expected to be built in hardware to conform to the MTRS concept.

2.1.1 Arithmetic operations

The different types of arithmetic operations supported by a processor are a large part of what makes it useful. Commercial processors typically have several variations of each basic operation, such as add and subtract, as well as separate versions for floating point and integer operations. These many versions of the same basic operation is necessary because of the different subtleties of each; one might be setting an overflow bit on overflow, another might not, and a third might be working on double-registers, and so on. This multitude allows a compiler to pick the one that is most suited to the task at hand, leading to more efficient code. Furthermore, it is common that many of the operations reside in the same functional unit, the so called ALU, and the instructions to access the different operations just change the operating mode of this functional unit.

As a proof-of-concept processor, we do not need all these different varieties to just execute code. We can make do with just one version of each basic arithmetic operations. Furthermore, each functional unit is supposedly fairly simple, and thus we would have one addition functional unit, one for subtraction, and so on. This does not mesh well with the omega network, as seen in section 2.1.2. So we limit ourselves to one type of arithmetic operator, namely addition, and only use one type of addition, namely unsigned addition without overflow checking. This particular addition method allows for use of integers greater then or equal to zero to add, but can be used with some care to also subtract fixed values from variables.

Having only an addition operation is sufficient for proving the MTRS concept, but is not really efficient as there are operations that cannot be performed, in particular bit-field operations such as and, or and bit shifts. An adder is however sufficient to emulate some of the higher order mathematical operations such as the already mentioned subtraction and also multiplications. Using the adder to make up a multiplier is hardly efficient compared to a hardware multiplier, but even so it is possible. The lack of some other commonly available operations limits the applications the prototype processor can be used for, thus reducing the viability of the prototype as a general-purpose processor. Even so, this is not a problem with MTRS, as the lack of functions can be amended in other versions of the design, and the functional units available can be chosen to cover a broader spectrum of available operations. It should be noted that this lack of functionality is not an uncommon state of affairs for prototype processors, the MIPS R1000 processor was also lacking several features, such as hardware division, which were not added until the R2000 design.

2.1.2 Multithreading

The MTRS design allows for many threads to execute at once. The actual number of threads is limited primarily by the interconnect fabric used in the processor, and in particular how many threads can be physically kept in latches at any one time. It is expected that with an appropriate instruction-to-bit-ratio the number of threads could be very large. This ratio is selected as part of the *Instruction Set Architecture* (ISA), and as such can be designed as required to the physical size of the processor.

The original design of the processor is based on an interconnect network using a sparse omega structure, and assuming that any instruction in flight is a separate thread. The benefits of this structure is as argued that it supports n functional units with a delay in time of at most $\log(n)$ clock cycles between them, and at the same time can support up for around $\frac{n}{2}(1 + \log(n))$ threads. The number of concurrent threads can go up to $n + n \log(n)$ if longer delay times can be accepted. The time it takes an instruction to move from one functional unit to another would on average increase as the number of threads increases, due to hardware contention for interconnect lines. The exact mathematical relationship can only be expressed as averages, as threads would interact in the omega structure to produce varying results. A complete mathematical analysis of this structure is outside the scope of the project, and as such has not been performed, but is certainly one of the ways in which the project can be expanded.

The reduced property of the omega network depends on the availability of duplicated functional units, and thus the amount of hardware spent on the interconnect network is directly related to the number of different functional units available in the processor. The omega network thus favours relatively complicated functional units, of which there are a small number of different types. Furthermore, a significant number of threads can be held in the interconnect network itself. It follows that the number of concurrent threads in the processor is related directly back to the number of functional units, the number of functional unit types, and the distribution of the functional unit types to functional units.

It was originally decided that there were going to be 16 functional units and 6 functional unit types. The number of actual functional units had to be a power of 2, due to the omega network structure. It would thus have been possible to build the processor using 8 functional units, but this was considered too small to explore the design concept properly. The functional unit types selected was *nop* for delaying execution, *load* and *store* for memory interaction, *cond* to evaluate conditionals, *add* to manipulate data, and *spawn* for thread and instruction management. These instructions are considered to cover enough of the operations needed in a processor to give a comprehensive evaluation of the MTRS concept.

The outline of the original interconnect network design can be seen in figure 2.1. As can be seen, the number of omega switches used in this network is 19, which is a reduction from the 32

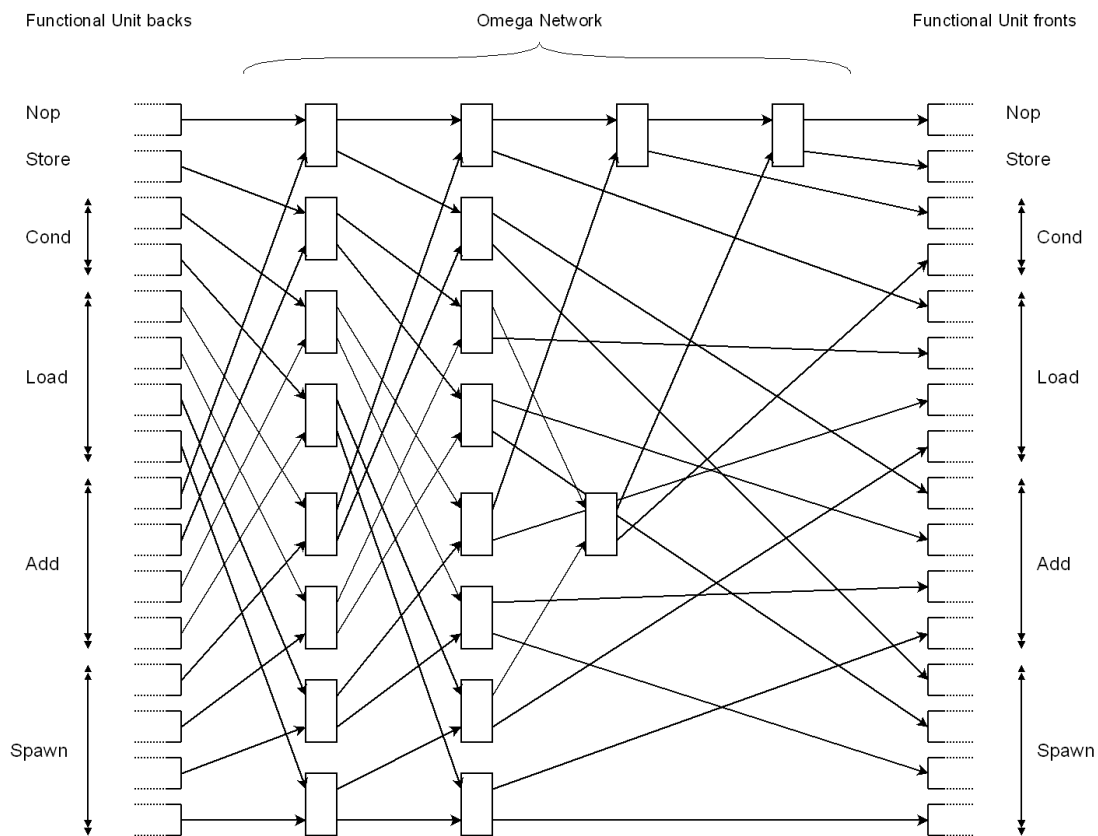


Figure 2.1: Diagram of the original omega network used for the interconnect.

that would have been required for a full omega network. This in turn translates to 19 instructions that can be held simultaneously in the interconnect network. Further, assuming that each functional unit can hold either two or three threads at once in a pipelined structure, we can see that the entire processor has a theoretical capacity of 59 simultaneous in-flight instructions, each from a separate thread. This is a somewhat conservative number due to the extra buffering capacity necessitated by the spawn functional units which is not included in the above calculations but even so it is a large number for a relatively limited processor prototype.

2.1.3 Register Size

The register size of a processor is an important design parameter, as it effectively determines how big numbers can be handled internally in the processor. Longer word sizes means that larger numbers can be handled, but also means that arithmetic operations take a longer time, which may lower the effective clock speed of the processor; longer registers also consume more silicon resources and so there are limits on how long registers can be used. While the MTRS concept does not generally use a register file, the considerations are the same; longer words will still consume more hardware resources. In addition, longer word sizes can be used as collections of shorter words by specialised functional units; a single 128-bit register might for example be used as 16 8-bit values to denote a 4 by 4 matrix, for example; then having several of those registers would

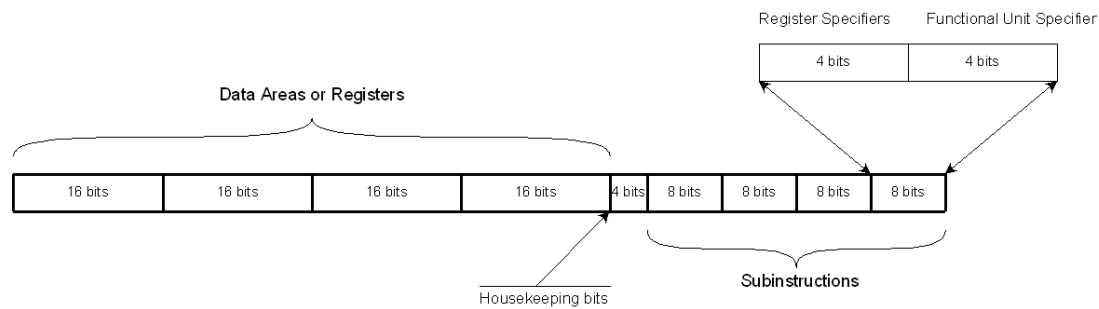


Figure 2.2: Diagram of the instruction format as initially intended.

not only allow for very large numbers but also native matrix operations, something that might be useful in scientific computing or computer graphics.

Instead of a register file, registers in MTRS are shipped around the processor attached to the instructions; thus, the length of the register values are directly influenced by the capacity of the interconnect network, and in particular of the width of the switches in the interconnect network. Thus the register size and number of registers are determined mostly by what can be fitted into the omega network connection width, which is limited.

For the prototype processor design, it was decided that the instructions would have four 16-bit registers to operate on, for a total of 64 bits wide. The omega paths are of limited width, and so cannot support too many registers, and a number of bits in the instruction have to be allocated to actual instruction data. It is possible to make eight 8-bit registers with the same number of bits, but this was decided to be too small a size for common operations such as delays, as an 8-bit register can only hold numbers less than 256. Similarly, having two 32-bit registers was not considered a good idea as having only two registers is excessively restrictive. Additionally, using four registers maps nicely to having four subinstructions in the instruction block.

2.1.4 Instruction size

There are supposed to be four subinstructions in an instruction, each with several register specifications and possibly other data. Due to the use of an omega network of size 16, each instruction uses four bits to designate the functional unit to transfer to next, as well as four bits to designate registers. This leads to each instruction being able to designate two registers each, as well as having an associated register based on its position in the instruction block, allowing three-register operations, while keeping each instruction to a limit of eight bits. With four instructions per block, this adds up to 32 bits for the instructions. An additional four bits is added on for internal processor housekeeping, keeping track of which subinstruction in the instruction block this is, for a total of 36 bits of instruction data.

The next subinstruction is designated with four bits each time, and spells out the exact functional unit that the instruction block should proceed to. A compiler might use this to designate

the exact path through the omega network, but this is redundant as the omega network only delivers to functional unit type instead of exact functional unit, due to the reduced property. Thus this value might as well be randomised or set to the first value of the block of identical functional units. The path initially chosen was to randomise this number in the compiler to point to one of the functional units for the function desired, and then let the omega network route as necessary to the correct functional unit connected to the originating functional unit.

It would be possible to compress some parts of the instructions, in particular the functional unit specifier, as there is redundant information there. The choice was made not to do this for the prototype, as this would necessitate extra decoding hardware and complicate matters, which would not be suitable for the prototype level of the design. It was felt that this could be considered for later designs if it was a problem. The clean relationship between instruction and functional unit was considered a good thing, as it possibly simplified writing assembly code for the processor.

2.1.5 Memory Manager

The memory interface in MTRS is supposed to handle a multitude of functions, but it is mostly outside the scope of the project. Instead of creating the full blown memory manager, the design has to settle with a less capable implementation. The ideal solution would be a minimal memory manager that is capable of supporting a real, physically separate, memory chip, possibly by reusing an existing memory interface design. It is expected that such a design together with a minimal MTRS memory manager might be able to handle the memory subsystem requirements.

An important issue for the initial design is whether to use direct addressed loads and stores or to use offset-modified addresses. Because of the instruction layout and the limit of 8 bits, 4 of which are already used for functional unit specification, the initial design decision was to use strictly directly addressed memory addresses. Both load and store instructions only have space for two register specifications of two bits each, as shown in figure 2.1, one holding an address and the other one being the register to load to or store from. Since there are no bits left to use for offset addressing, we cannot use this method.

It was also decided to address memory in 16-bit chunks, so that a single address points to the location of a 16-bit value. This is mainly to conform to the register width of the instructions, as having to use two memory accesses for each register load and store would be taking extra time, and using wider memory addresses would waste memory. Using 16-bit memory locations is the simplest choice given the 16-bit register sizes.

2.1.6 Initial Design Decision Summary

The design of the prototype processor uses an omega network connecting 16 functional units of 6 different types, doubling up some functional unit and having four identical copies of others. The

choice was further made to use four 16-bit registers, four 8-bit instruction words, and four book-keeping bits, for a total of 100 bits per instruction in flight. The omega network paths thus have to be 104 bits wide, taking into consideration the extrapolated four-bit omega addresses inserted internally in the network. These numbers are all trade offs between various design parameters, mostly related to the number of instructions available in the instruction set, the size of digits that can be used with the processor, and the size of these items in hardware.

Further, it is expected that we can use physical RAM to act as memory for the hardware implementation, with little overhead for a small memory manager. This together with the processor implementation is expected to fit in a commercial FPGA chip, and we should thus be able to simulate the hardware with some success.

2.2 Modified Design

Initial development of the processor hardware and simulations went ahead using the instruction layout shown in figure 2.2. The processor was completed to this specification in Verilog, and was as far as could be determined working as designed. Even so several problems arose when generating the assembler for this instruction format; the instruction format turned out to be not particularly suited to compiling and assembling code using automated tools. Some of the problems compiling to this specification are outlined in section 3.4.2 but are also covered in this section.

2.2.1 Instruction and Word Sizes

The first problem was that the instructions were not expressive enough, and used a third register reference for three-register operations. This third reference was as discussed before set to the numerical value of the subinstruction in the instruction block. When splitting code into instruction blocks, this feature frequently led to instruction blocks with a single operative subinstruction, and filling the rest of the instruction block with **nop** operations. This was not particularly efficient, and led to a disappointing bit to instruction ratio, as each instruction block as stored in RAM was 96 bits long.

The second problem was that MIPS used offset addressing a lot, as it has that capability. This can be emulated using an extra register and an **add** operation, but this is dependent on having that extra register. Having a free register for this purpose is not always practical, a fact made worse by the decision of only having four registers. Furthermore, freeing up a register for this purpose by spilling data to RAM is not possible, as it is the very action of reading or writing to RAM that needs the extra register. It was therefore clear that either the **add** instruction or the **store / load** instructions needed an intermediate value that was added in, specified in the instruction itself.

A further issue was that an instruction block that would lead on to another instruction block

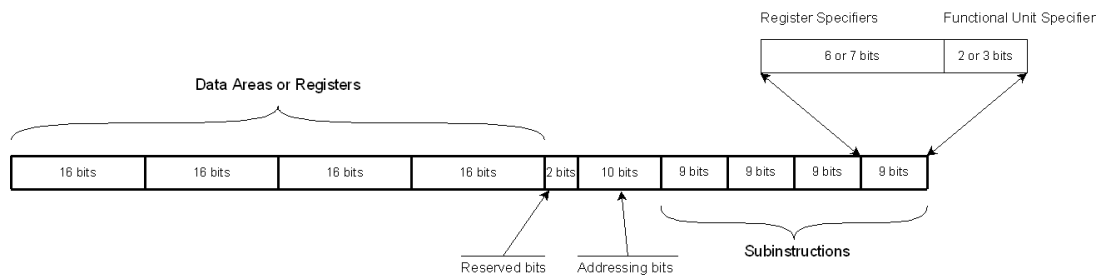


Figure 2.3: Diagram of the instruction format as eventually modified.

needed an explicit jump instruction at the end, which cut down the available useful instructions per instruction block to three instead of four. This again reduced the coding effectiveness and led to redundant code.

To solve these problems, another way of coding the subinstruction had to be found, but as the hardware implementation was already completed it could not be substantially different. Instead of reengineering the instruction set from the start, the existing set was altered so that compression of the subinstruction specifiers was used. This compression slightly increased the hardware complexity, but at the same time allowing a much lower compiler complexity and in particular a much less complex instruction set. Instead of the layout shown in figure 2.2, the new instruction block layout was specified as seen in figure 2.3.

Figure 2.3 shows how the instruction is laid out in RAM, with most significant bit to the left. The overall size of the instruction block has grown from 96 bits to 112, which is two whole bytes longer. As the memory interface is 16-bit, this is an extra memory reference to load instruction blocks, which was deemed acceptable as it solved the other problems associated with the instruction format. The reserved bits are currently unused, both in RAM and in the processor itself, but are included as they allow the 16-bit data values to be located on 16-bit boundaries in RAM.

As can be seen, the functional unit specifier was shortened to two or three bits, allocating a 2-bit coding to subinstructions that needed extra arguments. The subinstructions themselves were increased in size by one bit. This may seem trivial, but together with the compression of the functional unit specifier there was an extra either two or three bits of register specifier available in a subinstruction. The extra bits freed up were used to add in a three-bit numerical offset to load instructions, a two-bit numerical offset to store instructions, an extra register specifier to add instructions, making them take three specified registers as arguments, and an extra function specifier to the jump instructions. These changes thus addressed all the problems outlined with the original encoding.

The other change that was made was the introduction of the ten-bit address field into the instruction, replacing the requirement for a jump from each subinstruction. When an instruction block has run out of subinstructions, it now automatically gets an inserted jump which uses this address. This again used further hardware resources, but this was yet again balanced against the

more important consideration of code density. This field is ten-bit because this was the maximum size of the RAM available in the FPGA used. See section 2.2.2 for further details relating to memory size. If another RAM size is to be used, the number of bits allocated to this address may have to change.

One final change that was made to the instruction format was the addition of an operations specifier to the conditional subinstruction, making it distinguish between equal and not-equal conditions. The previous version of the instruction set had only supported one of those options, but it was found through the compiler that both versions were necessary. There is also space left in the instruction set to implement a less than and a greater than operation, but these operations were not realised in hardware due to hardware limitations. Greater than and less than can also be emulated in software given not-equal-to and addition, so their omission from the hardware does not imply that the hardware is inherently less capable.

The changes made in this second version of the instruction set were along the general idea of moving away from small and limited functional units to more capable functional units, that also take more arguments. Using this, one does not need as big an omega network, which is useful as there is a limit of the size of an omega network. It does however still cost in hardware as the omega network will need to transport larger instructions. The balance here between instruction size and omega network size is a topic that can provide for further investigation, which has not been carried out in this project.

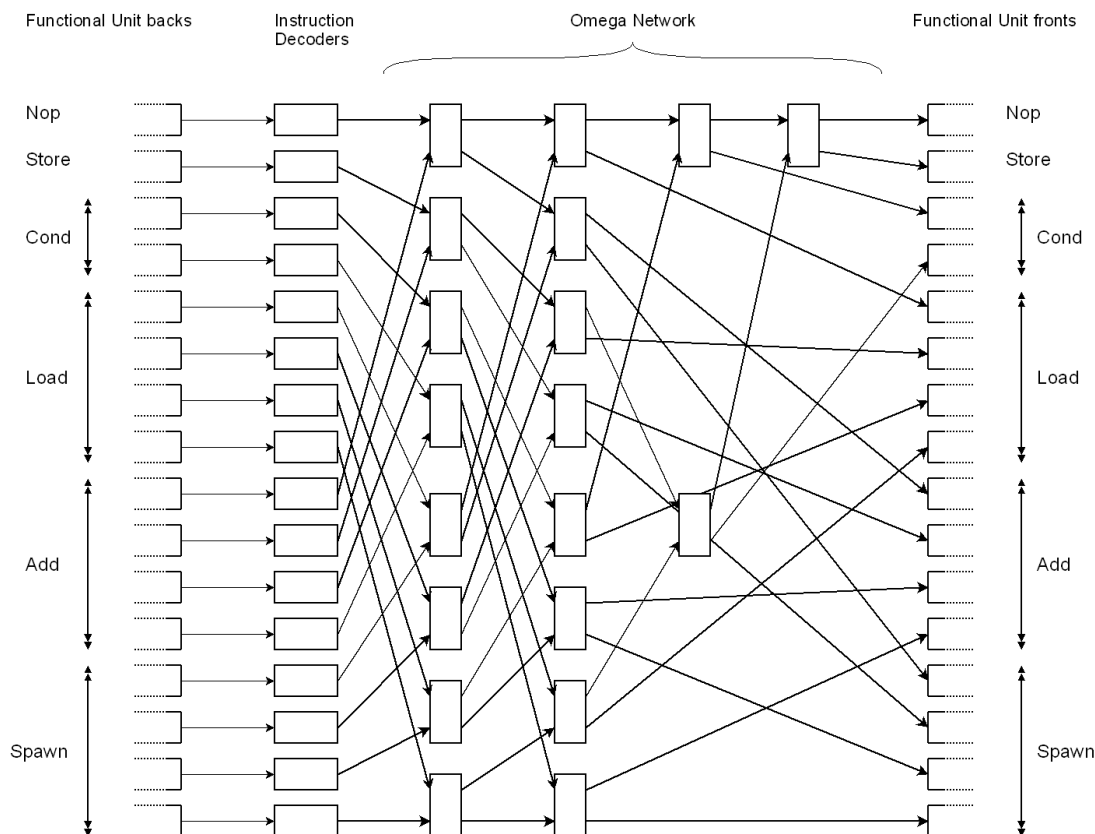


Figure 2.4: Diagram of omega network with instruction decoders added.

Figure 2.4 shows the omega network as implemented in the final prototype, with extra instruction decoder units added in before the first omega switches. This introduces an extra stage of delay through the network, making thread execution slower, but also increases the number of concurrently held instructions by 16, and thus increasing the maximum possible threads by the same number.

2.2.2 Memory Interface

It was found that it will not be possible to use real DRAM or SRAM, as their interfaces are too complicated to use. A hardware DRAM interface was found to take too much of the available space in the FPGA unit, and was also incompatible with the MTRS stateless concept unless a competent MTRS memory manager was also added, something that was outside the scope of the project. Therefore, another hardware solution had to be found.

The other problem that needs addressing is memory bandwidth. With two load units, one store unit, and four spawn units all interacting with memory, the memory bandwidth will need to be large enough to manage. Caches could be beneficial here as there might be as many as six concurrent memory reads and one memory write. There is however the problem that caches fare less than well in a multithreaded environment, as the threads memory access patterns are not linked and the basic premise of caches, the locality of data and instructions, is severely limited. The use of caches together with MTRS is a topic that requires further study, and the prototype implementation makes no use of caches.

The prototype implementation make use of the SRAM bundled in the FPGA unit the processor design is to be programmed into, and uses this as main memory rather than cache or buffers. The SRAM banks in FPGA units are neither overly numerous or particularly large, but with enough SRAM banks, the concurrent reads can all go ahead at once, as well as having each bank as a recipient for the store. This would mean having many SRAM banks contain the same data, cutting down on an already scarce resource. This is permissible for the prototype implementation which does not need a large memory, but does need it to conform to the memory interface necessary without creating a large memory management unit.

The prototype implementation has only one store unit, which means a maximum of one memory write in a cycle, so there is no problem with coherency of writes and a need for synchronisation due to writes to the same memory location in the same clock cycle; there is however an issue with reading from the same location in the same clock cycle it is written to. This problem has not been specifically addressed in the prototype implementation, instead it relies on the FPGA hardware SRAM to operate in a mode that resolve this difficulty by operating in a write-through mode, indicating that changed to memory locations are propagated if there is also a read to the same address in the same clock cycle.

Synchronisation errors can also occur when spawning a new microthread. Each microthread is assumed to be allocated its own frame pointer (see section 3.4.2 and 3.4.2), to hold thread local variables in RAM. The problem occurs if two or more threads are spawned in the same clock cycle, all of which needs a unique frame pointer, which is something that should be allocated by an operating system in conjunction with the MTRS MMU. Unfortunately there is no operating system and no MMU available for the prototype processor, but this problem still occurs. The only solution that was able to handle this situation was a memory-mapped hardware frame allocator, that knows about the different ways in which threads can be spawned and can generate four unique memory addresses in a single clock cycle on different communications lines. This unit can be considered the first part of the hardware MMU that is not included in the project. Programming assembler code for the MTRS system without this unit would be perfectly possible, but compiled programs with many microthreads cannot function without it and it was thus introduced, using yet more hardware resources.

2.2.3 Processor size

The prototype processor grew slightly with the introduction of these changes, but it was only of the order of ten or twenty percent. This however turned out to be enough to have it require a larger FPGA than was originally required. Originally the design was intended to fit into a Spartan 3E-500 FPGA from Xilinx, 3E being the type and 500 being a measure of size based on design gate equivalents, in this case 500,000 gates. Unfortunately this proved far too small, mostly due to the size of the omega network itself, and the design had to be housed inside a 3E-1600 instead, with an approximate equivalent gate count of 1,600,000. These size measurements are somewhat inexact, in that it is impossible to deduct an exact gate or transistor count without making a complete tape-out of the design, which was not attempted. There is some confidence in the gate-equivalent estimation however, and it is expected that an actual physical implementation of the completed prototype design should occupy a gate count within 20

As a comparison it should be noted that the Pentium 4 processor, on its first introduction to the market, had 42 million transistors and supported single-threaded execution, and that the most recent iteration of the dual-core Itanium, a VLIW architecture, has 1,700 million transistors and supports up to two concurrent threads; a current desktop processor design, the Core 2 Duo uses around 500 million transistors and again have up to two concurrent threads. While the prototype MTRS processor is lacking in many areas, notably number of instructions and lack of memory manager, it does support at least 50 concurrent microthreads while fitting into a number of gates that is a fraction of those for commercial processors. The transistor counts of these commercial processors also include on-die caches, which in the case of the Itanium is particularly large. As the MTRS prototype design does not have any caches added, the counts presented are skewed in

MTRS' favour. Also the MTRS is a prototype, and it is expected that a commercial design based on it would consume more transistors than the prototype. Even taking these factors into account, the MTRS design is at least on the same order of magnitude or smaller than current commercial processors as well as supporting many concurrent threads.

2.2.4 Summary

There were changes made to the original design of the instruction set and the processor, mainly because of problems encountered when creating the compiler for the design. These changes primarily added compression to the functional unit specifiers and an extra bit for the register specifier, but were enough to resolve the problems encountered. Some other changes, such as the addition of the next-address field, were added as the preceding changes allowed them to be and they addressed an existing problem, albeit one that could have been worked around in other ways. The changes made completely resolved the previous issues with creating a compiler, and can therefore be seen as successful.

2.3 Final Implementation

This section discusses the final implementation of the prototype processor and its subunits. In particular it covers how the final processor is configured and where it may be added to. The following discussion refers to the completed processor, the FPGA implementation of which have been shown to execute simple assembled programs with sufficient correctness that the output can be perfectly predicted.

The FPGA used for the physical implementation is a Xilinx Spartan 3E-1600, originally intended for embedded systems design in both hardware and software. It is mounted to a board with several IO options, most notably for our purpose driven serial ports LED indicators, but also on board Ethernet connectivity and various other possibilities. The board has two clock sources, one on 50 MHz and another at 66 MHz, either of which can be used for any purpose. The board also has a separate DRAM chip available to the FPGA, but this was not used due to the complexity of the memory manager needed. The prototype processor design utilised the serial port capabilities to load code into the processor and send output, but other than that used none of the IO-capabilities on the board itself. See section 3.3 for more information on the communications interface. The clock used was the 50 MHz clock, as the design compilation step advised that the highest overall possible clock speed the design could handle was less than 66 MHz.

There were some problems with fitting the complete design and a serial port sender (tx) and receiver (rx) unit into the FPGA itself, but these problems were eventually solved by minimizing logic and ensuring proper use of the FPGA capabilities. As for RAM, the prototype design ended

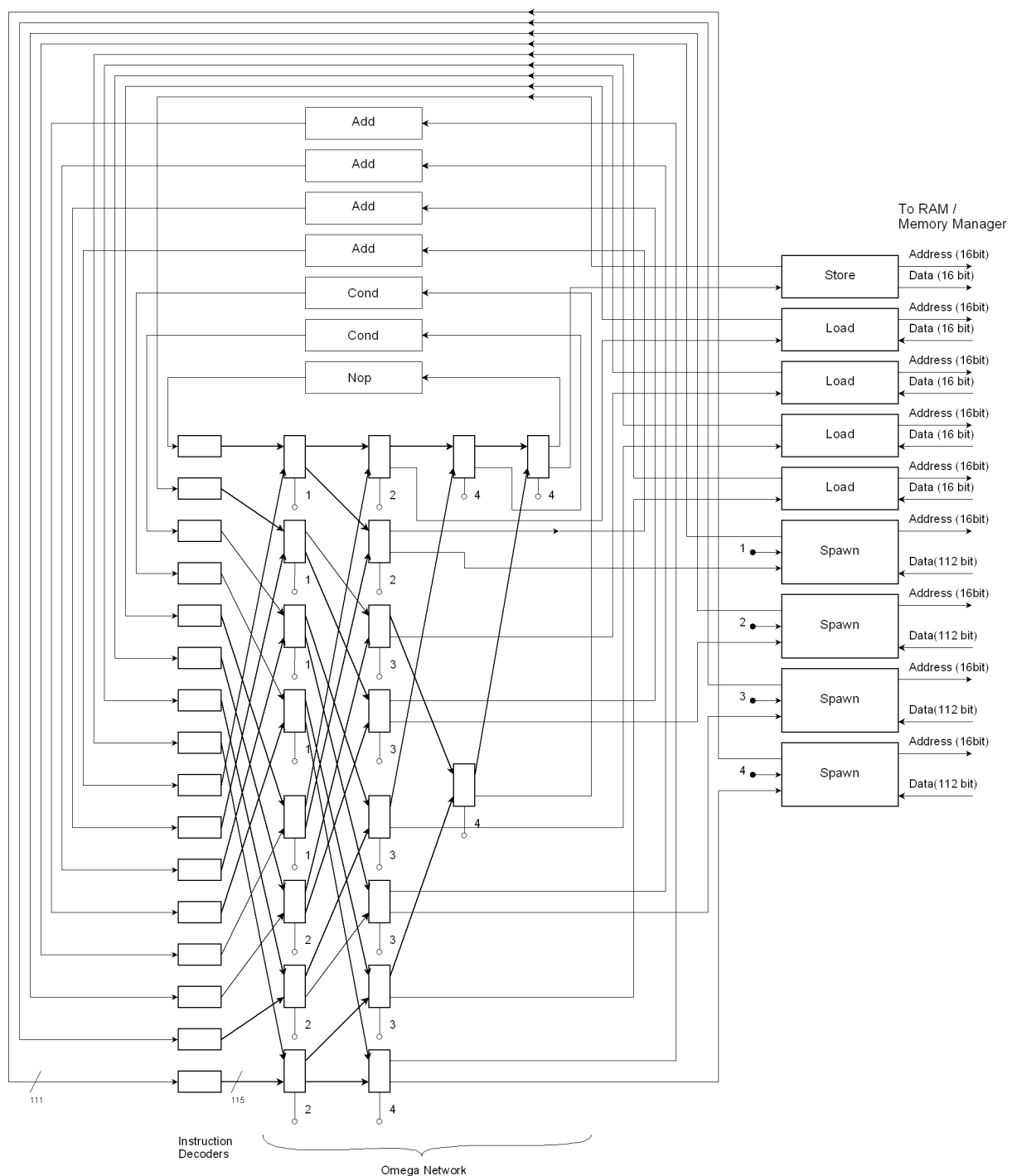


Figure 2.5: Diagram of the complete processor layout. Note the labels 1 through 4 that connect the omega switches directly to the spawn units and are each 111 bits wide.

up using 32 out of 36 available block RAMs in the FPGA itself as main memory instead of the DRAM on the board. The design thus fits fully into the FPGA itself, and can be programmed using the serial port. Figure 2.5 shows the completed prototype layout, not including the block RAM allocated and the serial IO interface. Note the not drawn but labelled wires 1 through 4 that connect the omega switches directly to the spawn units; these will be discussed further in section 2.3.8 below.

The functional units were all implemented as Verilog modules and integrated in a hierarchy.

At the highest level of this hierarchy was the FPGA module, which only had the physically available inputs and outputs defined; under this was defined the serial communications module, the processor module itself, and the RAM module, rather like a computer system in miniature. The processor module then incorporated all the functional unit modules, and one large module for the omega network. Several of the functional unit modules included child modules themselves, in some cases even children of children, to include the necessary functionality. The software used to compile Verilog to FPGA configuration was Xilinx Webpack, version 9.2.03i, which supported the FPGA chosen for the project. The Verilog code was written to be as general as possible and does not use any Xilinx particular extensions with the possible exception of block RAM definitions and the top-level Verilog module which by necessity have to be chip-specific.

2.3.1 Load Functional Unit

The load functional unit is intended to load 16-bit data values from the RAM into a register position. It does this by extracting the address from the register part of the instruction, adding it to the offset part from the instruction, sending the address to memory and waiting for the result, and finally, inserting the returned value into the correct register position. It does this in a fully pipelined fashion, so that it may operate on more than one instruction at once.

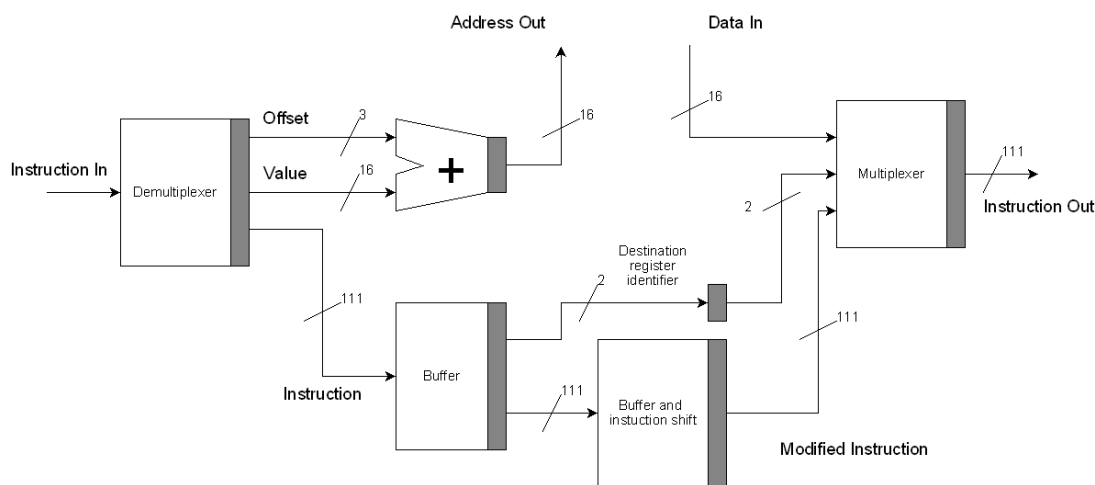


Figure 2.6: Diagram of the load functional unit.

Figure 2.6 shows a schematic diagram of the major subunits in the load functional unit. As can be seen, there are four stages in the pipeline for this unit and it can thus hold up to four simultaneous instructions. Only one instruction will be referred to memory at once, and memory is assumed to take one clock cycle to return the result. A longer but still fixed memory delay can be accommodated easily in this structure, but variable return times from memory would require a limited redesign of the buffer unit. A delay of one clock cycle is accurate for the FPGA on-chip block RAM used, and thus one clock cycle delay arrangement is sufficient for the prototype implementation.

The parts of the load functional unit that contain the important logic are the multiplexer and demultiplexer. The demultiplexer selects which register field gets put on to the address adder lines, as well as forwarding the directly coded offset. The rest of the instruction is forwarded straight to a buffer for a one cycle delay while the addition is performed. The multiplexer similarly takes the returned data from RAM and the destination register selector and inserts the data into the correct place, while forwarding the rest of the instruction untouched. This work is in both cases managed by a set of hardware multiplexers.

The only other logic performed is the forwarding of the instruction specifiers by nine bits. This is performed in the step marked instruction shift. This step forwards all register fields untouched, but moves the instruction specifiers by nine bits downwards, thus removing the load unit specifier off the bottom of the instruction block. This is in preparation for the next instruction decode, which looks at the bottom nine bits of the instruction block. Since this bit shift happens before the multiplexer, destination data needs to be preserved on two separate lines so that the returned data from RAM can be inserted in the right place even though the subinstruction bit-pattern has been replaced. This is clearly shown by the 'Destination register identifier' line shown in Figure 2.6.

There are four load functional units in the prototype processor, as loading data from memory to operate on is anticipated to be a much-needed operation, even including the spawn-induced automatic loads of variables. It is possible that there are more load units than are strictly needed, but this cannot be accurately determined without much further work on the MTRS prototype. Better compilation techniques for MTRS, which would lead to a significant number of programs compiled, could give an accurate average instruction count which is needed to evaluate the optimal counts of all the functional units included in the processor.

2.3.2 Store Functional Unit

The store unit has as its job to send data from the internal registers to RAM. This requires less complexity than the load unit as there is no need to coordinate any delays with the RAM response time. The only time-wise coordination needed is to make data and address appear to the RAM on the same clock cycle.

Figure 2.7 shows the layout of the logic of the store unit. As can be seen, this is again pipelined but this time only in two stages, allowing only two instructions at once. Extra circuitry in the demultiplexer ensures that both the address and data lines are set to all zero values if there is no instruction present, as well as invalidating the RAM write-enable signal. This allows for empty instruction in the pipeline. While these empty writes may not go through, for redundancy memory location zero is always set to zero, which is not a bad thing as it can then be used as a source for zero values as well as making the write-enable signal invalidation have an extra margin.

There is also no multiplexer in this unit in contrast to almost all other functional units. This

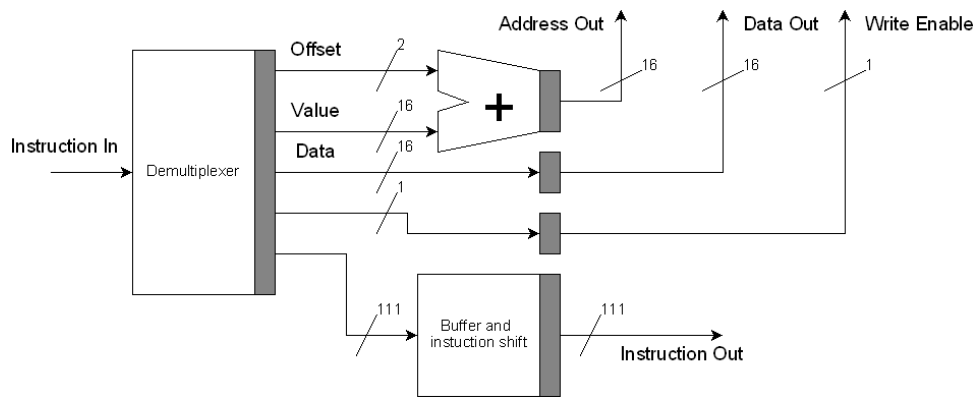


Figure 2.7: Diagram of the store functional unit.

is because there are no signals to multiplex together again; the store does not change any register values present but only forwards the subinstructions by nine bits, something that is done in the instruction shift subunit. Finished instructions can therefore go straight from this stage back into an instruction decode unit and on to the omega network.

The processor has only one store unit, which can be seen as a severe limitation. It is expected that the number of store units will have to be increased should the MTRS concept pass the prototype stage. If there are more than one store unit, extra circuitry will be needed to deal with concurrent writes to the same memory location, either by resolving it in some defined fashion or by supplying another method of synchronization between threads. This is one of the challenges that has not yet been resolved in the prototype implementation, and again because this functionality is supposedly handled by the memory manager.

2.3.3 Compare Functional Unit

The job of the conditional unit is to forward either one or two subinstructions in the instruction block, depending on the values of the registers specified to the instruction. It can either act as an is-equal or not-equal conditional, depending on an option is set in the instruction itself.

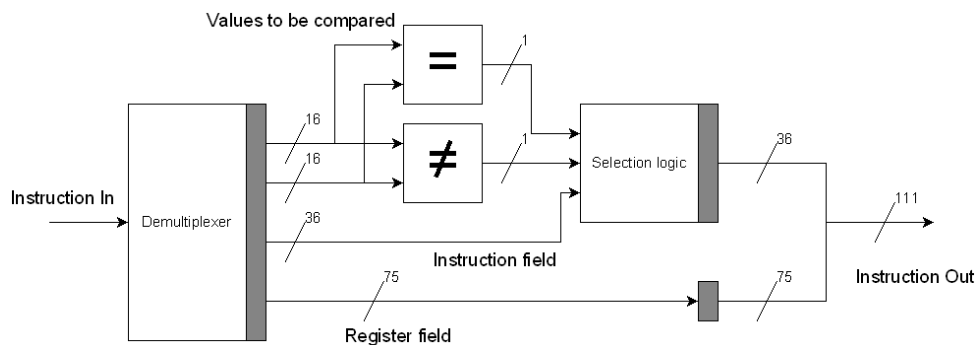


Figure 2.8: Diagram of the conditional functional unit.

Figure 2.8 shows an approximate schematic of the conditional unit. As can be seen this func-

tional unit is pipelined in two stages, first of which is a demultiplexer separating out the registers to operate on, the instruction and relevant control signals and the second stage doing the comparison work and selection logic. For equal and not-equal the comparisons are fairly straightforward and thus can be done in the same clock cycle as the instruction block forwarding logic. Again, there is an absence of a multiplexer at the end as there is no register value to multiplex back in.

The functional unit could be split into three stages fairly simply, but this was not done as it was not strictly necessary, and there were already problems with fitting the design into the FPGA. It is also possible to add a less than or greater than comparison fairly simply in parallel to the two compare units already present. Adding these extra comparisons would likely result in a third pipeline stage also being added in, and the functional unit would become much larger. To preserve space in the FPGA these features were omitted from the prototype design, but it should be a fairly simple operation to add them in as necessary, provided there is ample space in the physical instance of the design.

The unit as mentioned forwards the subinstructions in the instruction field by nine bits if the condition is met, and by 18 bits if the condition is not met. This is designed for a jump instruction in the space after the cond instruction, to divert instruction flow in case the condition is true. Thus a cond followed by a jump can be thought of as if-true-jump, if-not-true-continue. Of course, the sense can as always be inverted by using the complementary comparison function.

There are two conditional units in the processor, as the demand for conditionals are expected to be less than the demand for adders and load instructions. It is known that programs generally have a conditional on average every 6 to 8 instructions [2] and that the instructions between the conditionals vary in composition from program to program. Assuming that this remains a constant for MTRS, the lower number of conditional units to loads and add units is justified. Again, this is an assumption that is not yet backed up by empirical data, as the compiling techniques used are not yet complete enough to evaluate average instruction counts.

2.3.4 Add Functional Unit

The job of the add functional unit is to add two registers together and store the result in a third register. These registers can be all the same, or all different. This functional unit was discussed at length in the first part of the report, and its basic design has not changed.

As seen in Figure 2.9 the basic design of the adder is the same as in the first part of the report. It is pipelined into three stages, the first of which is the demultiplexer that separates out the registers to add and the destination register specifier. The second stage adds the registers together and shifts the instruction down by nine bits. The third stage multiplexes the result of the addition back into the instruction. As before the stages are independent, and the adder can thus operate on three simultaneous and non-related instructions, one in each stage.

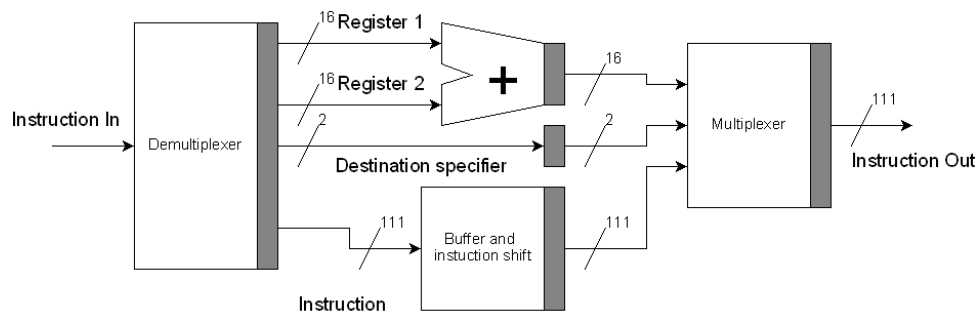


Figure 2.9: Diagram of the adder functional unit.

The adder here is unsigned, and no import is attached to any overflow that might be generated. Thus any addition operations that would result in a result larger than 65,536 will overflow silently. This property is not very useful for arbitrary integer arithmetic or even emulating arbitrary integer arithmetic. Various soft workarounds can be used to detect overflow, but will of course slow down any program using them as each addition needs to be checked. For this reason it is expected that should the MTRS concept develop further this adder unit may be extended with at least overflow detection capability, if not extended to a full ALU capable of addition and subtraction on signed, unsigned, and vector formats.

There are four adder units in the processor, as it is anticipated that add is one of the operations that a program will be utilizing to a large extent. In particular for mathematical applications this unit is expected to be heavily used, especially since it can be used to emulate a more capable multiplication unit in a microthread. Empirical data on how heavily this unit would be used in contrast to other mathematical functional units is again not possible at this time, and so the prototype processor has four adder functional units which can be used for additions, some limited subtraction, and for microthreaded multiplications.

2.3.5 Nop Functional Unit

The nop functional unit has two primary functions; to forward any instruction untouched except for shifting the instruction block down by nine bits, and secondly to clear spent instructions from the omega network. These functions are both accomplished in one clock cycle, and the functional unit is thus not pipelined at all. A diagram was left out at this point because there is nothing to make a diagram of.

The second function of the nop unit requires some explanation. Occasionally there will be instructions submitted to the omega network that have all four of its subinstructions used, and thus have 36 zero bits in the subinstruction field, see Figure 2.3. These instructions will be routed to the nop unit due to the '000' instruction specifier, and continue to be routed to nop forever unless they are taken out of circulation. The nop checks for these instructions and does not forward them back into the omega network, freeing up that instruction slot.

There is only one nop unit in the processor. While it could be argued that one could do without this, it is important to be able to do nothing in a thread every so often; furthermore, the cleaning out of instruction slots is an important function as without this the omega network may fill up with dead threads which do nothing but consume power.

2.3.6 Spawn Functional Unit

The spawn functional unit has as its purpose to execute any jump instruction and spawn instruction in a program. It also has as its function to handle the spills that are inevitably generated by the omega network. These two functions in concert lead to a fairly complicated functional unit which uses much logic circuitry.

Jump instructions and spawn instructions are collected together in the same functional unit because they share the same property; they both load a new instruction block from RAM. Whether this loaded block is then the only instruction block to be reinserted into the omega network or if both instructions are is what differentiates a jump from a spawn. In both cases there is the possibility of copying registers from the loader instruction to the newly loaded instruction, an action that is used extensively.

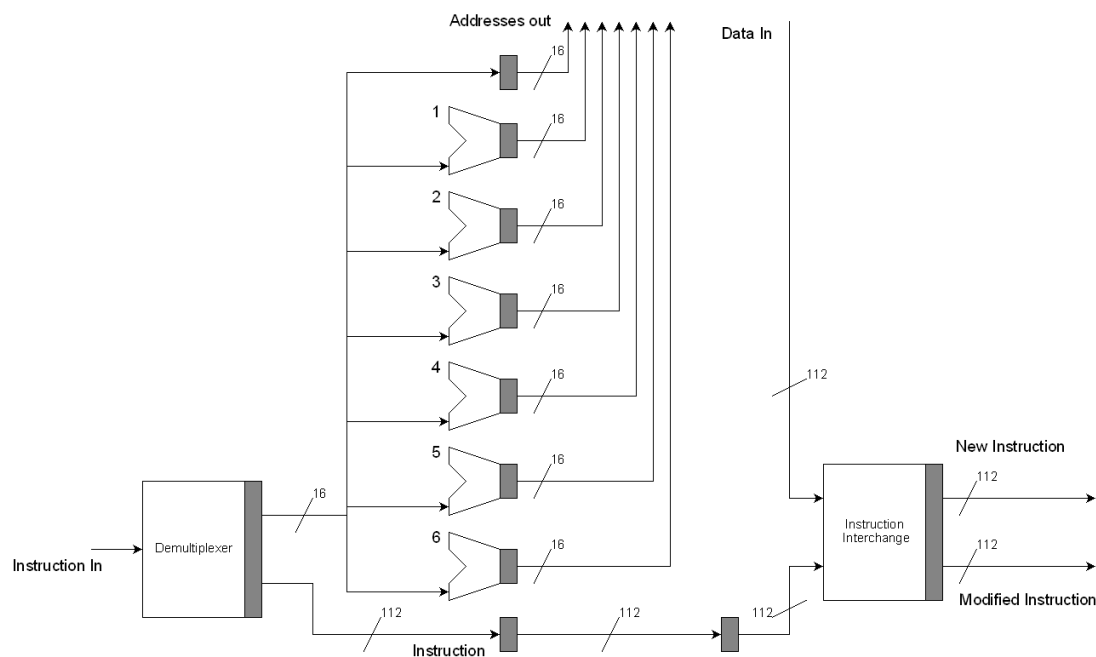


Figure 2.10: Diagram of the first part of the spawn functional unit.

Figure 2.10 shows the first part of the spawn functional unit as an approximate circuit schematic. As with all other functional units this is pipelined, this time in four stages. The first stage is the demultiplexer that separates out the address to load the new instruction from the rest of the instruction. The second stage is essentially a delay stage, the first of two, to let the RAM return the data; in addition the memory manager should here generate all the memory addresses necessary to load

the new 112 bit instruction in 16-bit chunks. This is shown in figure 2.10 by the six adders. These adders are logically associated with the RAM block in the prototype implementation, and should be located in the memory manager if we had one, but are shown here as part of the spawn unit for completeness. These adders generate the seven consecutive RAM addresses that the data is loaded from.

The third pipeline stage is again waiting, this time for the RAM to return the data. Again, as for the load functional unit, this can be extended if there is a memory management unit or if there is slower RAM. One clock cycle delay is enough for the on-FPGA block RAMs.

The last stage is the most complicated since it intermingles the loaded data and the registers from the issuing instruction. It is also here the issuing instruction is blocked if the instruction is a jump or allowed to go forward if it is a spawn instruction. Which registers are copied to the new instruction from the old is controlled by the issuing instruction, see section A.1.3.2.

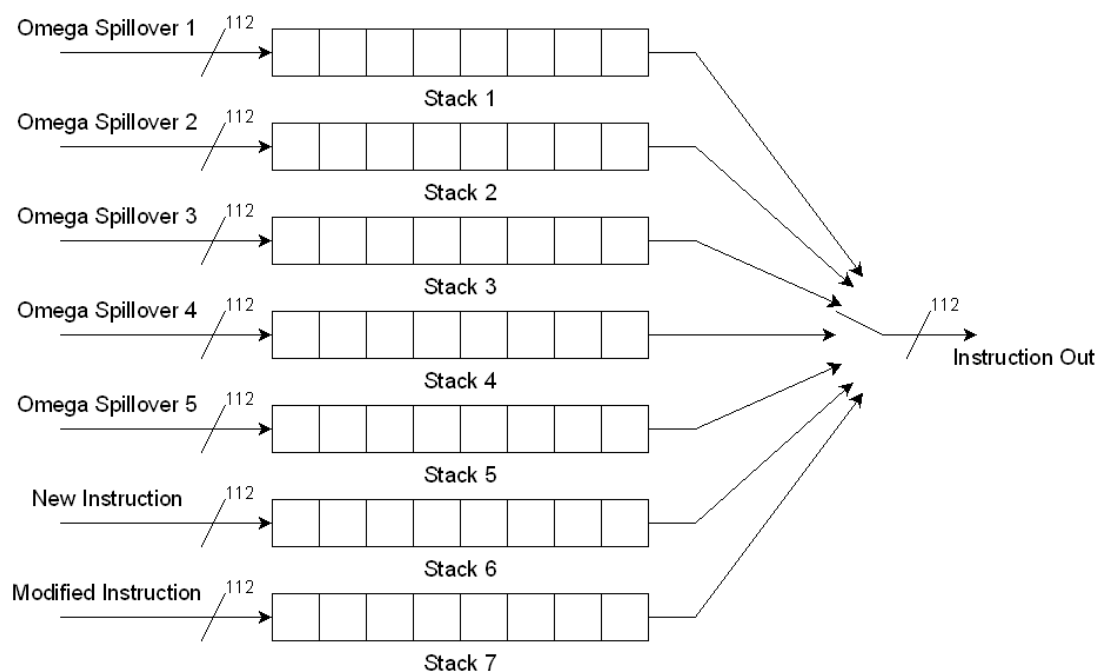


Figure 2.11: Conceptual diagram of the second part of the spawn functional unit. The selector is driven by the fill levels of the stacks.

The output from this initial stage is then forwarded to the second half of the spawn unit, which does not interact with RAM but instead has as its sole purpose to schedule instructions to insert into the omega network. A schematic representation of this can be seen in figure 2.11. Each incoming instruction is added to a hardware stack eight elements deep, and the next output is selected from the fullest stack. In addition, if there was an output and input to the same stack in the same clock cycle, the instruction in question is immediately output instead. This rather complex arrangement is necessary as the omega network tends to drop instructions at random and there needs to be a buffer to reinsert them. The rate at which instructions are dropped in relation to the number of concurrent threads is not determined, but one could use markov chains to predict such

a value. This was not done for this project, and the stack size of eight is mainly set due to limited hardware. Continuing the project, an instruction drop out value should preferably be determined, and the buffers sized accordingly. As it is there are 152 buffer spaces available in total, which is a much larger value than the theoretical maximum number of threads in the processor, and it is expected that this value will be sufficient for any proof-of-concept programs.

These stacks and the related logic are complicated, and thus takes a log of hardware resources. In addition, there are four spawn units in the prototype processor. It is expected that a spawn unit can insert a single instruction into the omega network on each clock cycle, and furthermore that instructions take on average 16 clock cycles to return to a spawn unit; thus these four spawn units can have 64 instructions in flight at once. It can clearly be seen that this is in line with the predicted number of maximum threads of the prototype processor; furthermore, if there were only two spawn units this number of threads could not be met and the prototype would have excessive idle times. Four spawn units is therefore the right number for this size of MTRS processor, despite their complexity as less units would not be able to keep the MTRS working at full capacity.

2.3.7 Instruction Decode

The instruction decode unit has as its job to interpret the shortened instruction specifiers, and to append the four routing bits to the instruction so that it is transported correctly through the omega network. This is in general not a difficult problem, and thus is completed in one clock cycle.

These units are mandated by the changes made to free up bits in instructions, as outlined in section 2.2. The functional unit that these units route instructions to is always the first one of the specified type; so it only explicitly routes to the first cond, add, load, and spawn. This explicit routing does not matter as the instruction is forwarded to a functional unit depending on which functional unit is actually connected rather than the explicitly named one; this is a property of the reduced omega network. For example, instructions passed through spawn unit number three and its subsequent instruction decoder, destined for another spawn unit will have the address of spawn unit one written into the routing bits. This will route it to spawn unit three again, as this is the only spawn unit actually directly connected to spawn unit three as indicated in figure 2.4.

To write the correct routing information each instruction decoder has to have an idea of its place in the order of instruction decode units, and thus each needs to be different in hardware. Rather than creating 16 nearly identical instruction decoder modules, only one was made but it was arranged to take the argument of its location, which was then set to a fixed value in the next Verilog module up in the hierarchy. This fixed value may or may not be used by the Verilog compiler to create 16 different units from one.

2.3.8 Interconnect Network

The interconnect network is as already discussed the reduced omega network, which routes instructions to the next functional units. The complication is that at each omega switch in the network, an instruction may be dropped out due to contention, as each output connection can only handle one instruction in any given clock cycle. This problem could be solved by internal buffering in the omega network, but it was felt that this would be too complex on an already complicated switch, and instead the instructions were dropped right away and forwarded into a spawn unit. Each spawn unit is capable of handling up to five such spill instructions in a clock cycle, see 2.3.6 for more on this. There are nineteen omega switches in the omega network, which lead to one spill instruction slot being left over, this was later used by the program loader (see section 3.3) to load programs into the prototype design.

The reduced omega network is otherwise as it was described in part 1 of the project report. Due to the drop out and associated delay there is however a chance that an instruction will take longer than $\log(n)$ to traverse the network for that instruction; thus the average time through the network is lower. The chance of this happening and thus the overall throughput is dependent on the exact instructions in flight and thus on the programs active. The exact delay can therefore not be determined by other than experimental results.

2.3.9 Serial Output

The prototype processor is able to send data to the host computer over a RS-232 connector on the FPGA board, which is mapped directly to pins on the FPGA itself. RS-232 was used for all communications as it was available on the prototype board and is relatively uncomplicated compared to other options. Sending data takes the form of sending double-bytes, i.e. two bytes at a time, to map with the internal 16-bit register and memory sizes. The speed of the RS-232 connection used for the prototype processor is always set to 115200 baud, with eight bits, null parity, and two stop bits. Transmitting an entire double byte thus takes 20.83 ms. Comparing this to the FPGA clock frequency of 50 MHz, it is clear there are about 10^4 processor clock cycles between the times we can issue a single double byte output write.

Data that is to be sent is written to memory address 0xF000, which is outside the range of the internal memory. This memory address can also be read from, but the result will always be 0x0000. Also note that data written to this memory location is sent immediately, unless there is another data value being sent already. There is no way of detecting another data value being sent, and from the above discussion of comparative clock rates it is clear that we have to wait for output operations to complete. Thus it is likely that output from the prototype processor would necessitate a separate thread to ensure timing of sending characters.

2.3.10 Summary

The prototype design is comprised of a hierarchy of Verilog code and implemented in a FPGA. The implementation is complete in that all functional units are present and working, and that code can be executed on the prototype. It is incomplete in the sense that it is not yet a general processor, the memory management unit is still missing, and further refinements can be made to much of the processor.

The largest current problem with the design is the lack of a memory management unit and the lack of support for many of the common operations necessary in a processor, in particular bitwise operations. There is however space available for such extensions either by increasing the size of the omega network or preferably by making functional units multipurpose.

2.4 Testing

Testing of the processor was primarily carried out by tests on the component functional units separately, but there were also full-system simulations carried out. These tests combined to give enough coverage to locate several problems or bugs in the implementation. Even so they could not cover every aspect and interaction in the MTRS prototype, and it is expected that the prototype will still have bugs that have not been found. The testing does however give confidence that remaining bugs are not severe enough to invalidate the MTRS concept.

2.4.1 Verilog Unit Tests

For each functional unit completed, a unit test was applied specifically to that Verilog module, and then simulated to give confidence that the unit fulfilled its goals. In most cases problems were uncovered in the functional units, and corrected. These problems were mostly logic errors, which required a small amount of corrections to fix.

Verilog tests consisted of predesigned inputs that were intended to produce specific outputs from the unit under test. The number of different inputs, and thus test points, used depended on the functional unit under test. The spawn unit had the largest number of individual tests as it was the most complex functional unit, while units like the adder had only a few test points confirming that it was working. The output from the tests were manually inspected as opposed to automatically checked, as automatic testing would take longer to set up.

These unit tests uncovered logic errors in the functional units themselves, which were in themselves fairly minor but would have been difficult to locate in the completed processor. The testing methodology also allowed these errors to be found before the full system was put together, and thus lend a degree of confidence to the system as a whole even before system testing.

2.4.2 Processor Simulations

The Verilog tests were used with the simulation tools build into the Xilinx software used for the project, as this allowed inspection of arbitrary internal signals of the unit under test. The processor was simulated using these tools to find problems in the interactions between the functional units. These simulations were fairly time consuming due to the large number of games that needed to be simulated, but gave accurate information of the working of the processor.

Problems that were unearthed were of the order of a functional unit not operating with the other functional units for some reason; for example by not forwarding the correct bits in the instruction word as it should. These errors were found by sending test instructions into the processor and simulating while manually following them through.

As with the previous simulations, the output was manually inspected for problems and errors. This was more difficult with the larger simulations as there were many internal signals in the processor that needed to be inspected due to the complexity of the processor. Even so it was facilitated by following instructions through the pipeline and looking at single points in time when the instruction appeared on a small subset of the wires, and remained a feasible task.

2.4.3 Full-system Software Simulations

The full system simulations simulated the entire processor as well as the connected units, namely the RAM and the serial input/output module, together. These simulations were slower and more detailed than the processor simulations, but gave insights on how the system worked as a whole. It was even possible to simulate a full instruction being loaded from RAM and executing, and in turn loading further instruction blocks. This effectively simulated the entire prototype operating, which was most useful for finding issues in the completed processor.

These simulations were used to debug problems with the entire hardware assembly, and helped show that the prototype processor implementation was executing programs correctly. The downside to the full system simulations was their long runtime, which was of the order of hours on a reasonably fast workstation. Using these simulations, test instructions were loaded into the processor through the serial interface and writing things to RAM was tested. Among the problems found and corrected by the full-system simulations were problems with transferring instructions to the processor through the serial port and receiving the output.

The major outcome of the full-system simulations was confidence in the serial connection hardware, which was important as any software loaded into the processor is transferred through that interface. The tests were thorough enough that when the actual FPGA hardware was then connected up and code was executed on it, there were no problems that could be attested to the hardware executing the code incorrectly.

2.5 Summary

The MTRS prototype processor was successfully implemented in a FPGA package, and can be shown to execute instructions correctly. The processor is comprised of sixteen functional units connected by an omega network, which embodies the design goals set out in the project goals and introduction section, foremost reduced state and microthreading. These goals have been met in the prototype implementation, and it is shown that is possible to construct a processor designed along these principles.

The processor was initially specified through the design criteria and the use of the omega network to fulfil them, and the functional units added to perform certain functions. Eventually there was a host of problems with the initial implementation, and certain design choices made had to be changed, most notably the instruction coding in the instruction set architecture. None of the changes affected the underlying MTRS concept, but were rather practical adaptations and solutions to problems that had occurred in the original implementation. The solutions were in every case successful in resolving the encountered issues.

Registers in the MTRS processor are not kept in a register file as is normal for microprocessors, but are instead kept together with the instructions in an instruction block. This is to reduce the state of a processor, as the state in a register file is inherently associated with the processor and not the process, thus keeping with the reduced state goal. Despite this, the data fields are still referred to as registers as they behave as a register file from the point of view of the ISA.

The final implementation of the prototype processor uses 16-bit registers and nine-bit instructions, all packed together in 48-bit to 112-bit instruction blocks. The instruction blocks are loaded from memory, either as a new microthread or as the continuation of an existing microthread, and are executed by traversing the omega network and visiting the functional units. The processor can manage the execution of at least 50 such instruction blocks at once, thus fulfilling the microthreaded design goal.

The processor, when viewed from the software side, has no inherent state in itself, but each microthread contains its own thread state. The processor can still be modelled as an extremely large state machine, but it has very few logic interaction points between the separate instruction paths. This succeeds in the reduced state goal, as each thread is now not tied to processor state.

The processor is not yet general enough to be a general purpose processor, mainly due to the lack of certain features, such as a completed Memory management unit and bit-shift operators. The goals of the project were not to create a completed general purpose processor, but to prove the viability and construct a reduced state microthreaded processor. These goals have been completely fulfilled as the processor is tested to be functional and is even implemented in a FPGA fabric.

3 Software Tools

3.1 Introduction

The software toolchain for this project consists of four main parts. In order from more to less hardware dependency these are the Loader, the Assembler, the Linker and the Compiler. They all fit together serially as shown in figure 3.1, and are dependant on each other in order to compile code for the MTRS platform. The Loader is tasked with transferring a binary memory image containing executable source code to the physical SRAM of the processor, and have parts implemented in both software and hardware. The assembler has as its job to create the executable binary memory image from a list of assembly-level instructions. The assembly process also include resolving variables and instructions to absolute numerical addresses. The linker is tasked with creating the full list of instructions, given a shorter list of instructions that refer other named lists of instructions; this can also be viewed as inserting library functions into the assembly code. The compiler finally has as its job to create the assembly-level listings that the linker uses from higher-level source code descriptions. The whole forms a toolchain from high-level languages to binary executables loaded into the RAM of the physical implementation.

3.2 Assembler

The function of an assembler is to transform a human-readable list of instructions to a machine-executable list of instructions, that is, to make a program executable. Assemblers traditionally do a little bit more than this; some limited built in macro facilities and some simple numerical expression evaluation are usually included. While these features are not required they are usually useful for assembly programmers and are included to speed developer productivity.

The MTRS assembler will not have any extra features other than translating human-readable instruction description into binary instruction descriptions. This is because it is not intended as a programmer tool but as a proof of concept assembler to show that the MTRS programs can assemble, and that the resultant code can be executed on the MTRS prototype processor. The assembly process itself is done on a host machine, as the MTRS system is not yet mature enough to host programs of this complexity.

In the overall scheme, the assembler fits in between the compiler and the loader software, as can be seen in figure 3.1. The linker is usually integrated into the assembler, or done after the assembler is finished; it is for various reasons elaborated on later done before the assembler in this toolchain. This section first goes over the languages the MTRS assembler will have to translate between, and then goes into details of how the translation is performed.

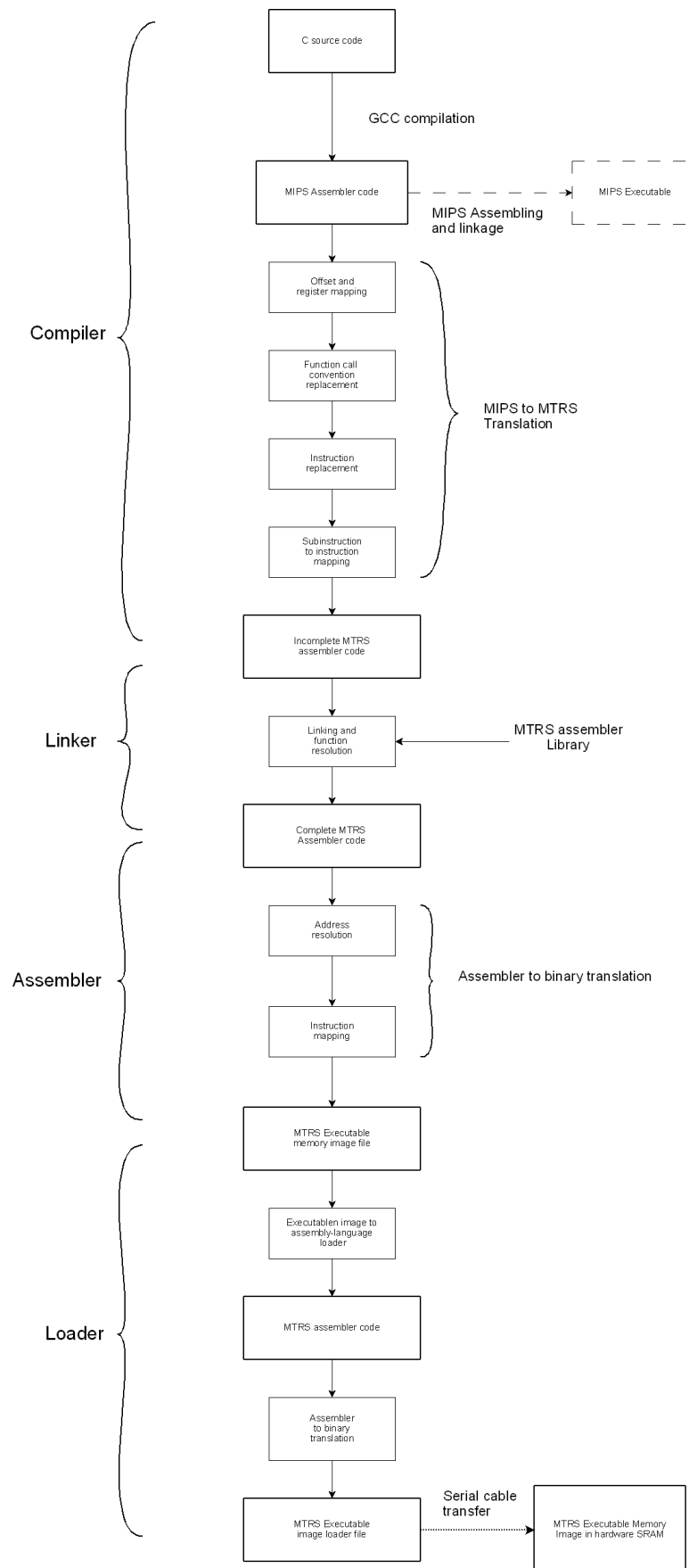


Figure 3.1: Schematic of the complete software toolchain created to compile code for the MTRS processor.

3.2.1 Motivation

The assembler is required to transform assembly-level code into actual binary codes that are executable by the MTRS processor. This is needed as the processor does not interpret ASCII coded data but uses bit patterns directly, and the ASCII coded instructions must therefore be translated. While this could be done in hardware, it is too complicated a process, and it is only applied to each program once. Thus, the assembly is done in software on a host machine and then transferred to the MTRS processor.

Without an assembler there is no efficient way of showing that the MTRS prototype does work as intended, nor that it can be programmed using high-level languages. It would be possible to craft binary programs directly, by hand, and transfer them to the MTRS prototype to be executed; but this would be slow and error-prone. The assembler offers a cleaner way of showing that the prototype have the required functionality.

3.2.2 Language Choices

The assembler language is discussed in detail in appendix A.1, and is not covered fully here; this section covers design choices and particular features that are important and that differentiate this assembler language from others, as well as the choices that resulted in the format of the assembler language.

The assembler language needed to fulfil two criteria; it needed to be detailed enough to map exactly to the binary instructions used by the processor, but it also needed to be human readable and writable, which meant using a subset of natural language. This tradeoff is nothing new, and has been done for all assembler languages created. The particular problem with the MTRS instructions is their grouping into instruction blocks. This has to either be expressed in the assembler language or be implicitly added in the assembler itself. It was decided that the breaking into instruction blocks were going to be accomplished explicitly in the assembler language itself, as this allowed the most flexibility; having the assembler break the instruction stream automatically would possibly have led to misaligned and therefore slower code. Making the blocks explicit allows the compiler to schedule instructions together and the possibility of creating faster code. The `.CALL` assembler sequence was used to initiate an instruction block, and it is required that a `.CALL` instruction is present at least every four instructions in assembler code. Section A.1.3.1 contains information on how the `.CALL` instruction is used.

The other features of the binary instructions needs also to be featured, particularly the register loads from RAM implicit in instruction loads. These are represented in the assembler language as `.SET` instructions. In analogy with the binary instruction formats, there can be up to four `.SET` directives in an instruction block, each corresponding to one of the registers loaded. In the binary

representation, the register data comes after the instructions; but in the assembler they may be intermingled as necessary. The register data will still be loaded at the start of the instruction block, so will overwrite registers at that point regardless of where in the instruction block they are located in the assembler. The syntax of the `.SET` instruction is discussed further in section A.1.3.1.

The binary representation of instructions have both the jump and spawn functionality mapped to the same functional unit as if they were the same instruction. They do however have very different purpose even though their mode of operation is very similar. Because of this disparity between their function and their implementation these instructions been separated into two different instructions in the assembler language, but still map to the same instruction in the binary representation. The instructions are naturally called **jump** and **spawn** respectively, and take very similar arguments in the assembler representation. Section A.1.3.2 have more information on the syntax of these instructions.

3.2.3 Implementation

The assembler and linker was implemented as two relatively independent but communicating programs. The linker takes compiled code and makes it into complete assembler code, while the assembler proper assembles the output from the linker into the binary MTRS format. The programs communicate in one direction only, from the linker to the assembler, and use normal files as their communications medium. The following sections contains relatively low level implementation details on both the linker and the assembler.

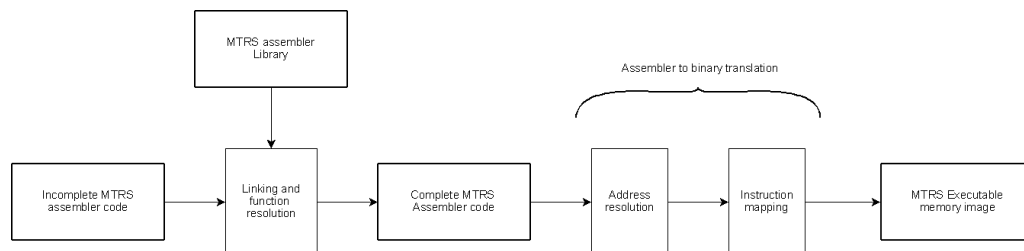


Figure 3.2: Schematic of the linking and assembly process.

Figure 3.2 shows how the assembler part of the toolchain fits together in detail. The separate steps in this figure are few, but even so the changes made are fairly substantial. It is the language design of the assembly language that allows the steps to be so few, in that the assembly language can in general be translated directly.

Linker Implementation

The MTRS linker works on assembler level as opposed to the binary level. The reason for this is that it is easier to manipulate string data then binary data on a conceptual level. There is no technical reason the linker could not work on the binary output of the assembler, given proper data

is included in the assembled file. The reason the linking is done on the assembler language level is simply that it is easier to debug and test the assembler if it is working on human-readable data.

The linker first splits the input assembler file into its constituent functions, creating a separate file for each one, and keeping track of what subroutine name is in which file. It then starts with the top level function present in the original assembler file, copies it to a final output file, and records what functions it refers to as well what function name has just been included. It then repeats this process for each function that has been referred to but has not yet been included. Whether a function being referred to is in an available assembler function library or was already present in the original file is irrelevant, both sources are searched for a correctly named function. This ensures that library functions are included as long as they are uniquely named and referred to by that name in the assembler code passed to the linker. Should the linker come across a function name that it does not recognise, it cannot go on and will stop there.

The final file is appended with any function that comes up in this width first search of function names. Once there are no more functions to include in the final file, the linker is finished and the final file can be passed to the assembler itself as it now has no unresolved function references.

The relatively simple linking process have some general properties that would need to be amended if the MTRS concept is to be taken further. The main unaddressed feature is that there is no method to refer to already present libraries on the target system, and no way to generate binary libraries and load them into a particular address. This is a critical feature, as each program compiled with this linker has a static copy of the functions it uses included. This is bad for memory size usage, and particularly so for a multithreaded system. This will need to be addressed going forward to ensure the memory usage for the MTRS architecture is not unreasonable.

Assembler Implementation

The assembler is a relatively simple program, the only point of difficulty is implementing it on an 8-bit native machine when instructions are coded in 9-bit blocks. Apart from this difficulty, it is fairly straightforward to translate the assembler language to MTRS prototype binary representation. The input to the assembler can either be a completely linked file fed from the linker or a hand-crafted assembler file, or even an already assembled file that has been recomposed into assembler statements. For a use of this last function, see section 3.3. Regardless of its origins, the assembler-language file is processed the same way.

The current assembler implementation uses a two-pass approach to mapping the assembler language; the first pass over the assembler file collects the instruction block labels and resolves at what address they will appear in the final MTRS binary image. This is dependant on how many registers are loaded in that particular instruction block. An instruction block can thus vary in size from 48 bits to the full 112 bits in increments of 16. The address counter is aware of this and uses

it to compute the address the next instruction block will appear at. It then maps each instruction block label to an address, and stores this information for use in the second pass.

The second pass in turn starts at the top of the assembler file it is given and translates each instruction line and block in order, and inserts the resultant binary data into the output file. Due to translating each line in order the assembler have no grasp of the overall structure of the program it is assembling, but just translates one form of the instruction specification into another. The difficulty is as mentioned above to output correct 9-bit sequences on systems that are natively eight or 16 bit. This is however only a matter of programming the assembler correctly, and is not a limitation in itself, as such the problem was eventually resolved. The only other particular in this stage is that instructions referring to labels have the labels replaced by the numerical addresses computed in the previous pass, and so are completely fixed. This only applies to instructions that reference memory, and due to the first step resolving these addresses, is fairly simple.

The result of the assembler is then a binary file that is a perfect image of the contents of the MTRS prototypes RAM when it should begin execution. The assembler automatically include the relevant start offset needed, and locates the first instruction of the top level function at address 0x0002. Should the program prove too large to fit into the available prototype RAM, the assembler will fail with an error; the amount of RAM available in the prototype is a variable in the current assembler but its value is fairly easy to change. The program size check can even be ignored if needed, for example for generating program loader code as seen in section 3.3.

3.2.4 Testing

The assembler and linker were tested with a small set of programs, mainly the ones that were subsequently used to program the physical prototype or were generated from higher-level code by the compiler. Testing was in general performed by using the assembler on these programs and then inspecting the output, and comparing it to the expected output. The generated binary output code was inspected through a hex-viewer and compared to what it was expected to be from assembling the code by hand. This process was continued until subsequent modifications to the assembler made the results correspond for each instruction. As the assembler is a fairly simple translator, this in general worked well and resolved a number of bugs in the assembler to the point where the assembler is currently trusted to generate the correct code. The code generated by the assembler has also been run on the MTRS prototype itself giving further indication that the assembler is producing correct output.

The same testing strategy was used with the linker, but the checking here was simpler as both input and output data could be inspected directly. The simple algorithm used for the linker made this further simple as there were not many points the process could fail. The linker is therefore also considered complete and relatively bug free.

3.2.5 Summary

The linker and assembler are integral parts of the MTRS toolchain, which is used to program the MTRS prototype. Code that is being compiled for the processor prototype is fed through first the linker which resolved all unknown functions and inserts them, and then through the assembler which resolved the addresses of the functions and translates the whole file into an binary representation which is a bit-for-bit image of what is expected to appear in the RAM of the MTRS prototype processor that has been developed.

The assembler and linker have one major problem in their current implementation, namely that they only use statically inserted code. There is no provisions whatsoever for dynamically loaded code, which will become a problem as the MTRS system is intended to be very multithreaded. Having only static code will make the multithreading very memory intensive, and is not recommended. It is therefore necessary to implement dynamic loading of functions should the MTRS project go forward.

Even with this conceptual problem present, the assembler and linker as written are reliable enough to produce proof of concept code for the MTRS prototype. It is possible to write code in the assembler language created and have it execute correctly on the processor prototype.

3.3 Program Loader

The program loader is responsible for taking an assembled MTRS file and transfer it to the MTRS hardware. This is accomplished through the use of standard RS-232 communications that are available both on the host and the FPGA board. The RS-232 module in the FPGA is connected through a redundant spillover port (see 2.3.8) and can thus inject instructions straight into the interconnect network of the processor. The software side encapsulates the desired program into a specific format for transferring over the serial cable which is then unpacked again by MTRS software and stored in block RAM.

3.3.1 Hardware

Serial communications were used for their relative simplicity and their broad interoperability. Using either a USB or Ethernet communications method would have been more complex, in particular in hardware used, and thus put too large a demand on the logic space available in the FPGA. It is possible that either USB connections or Ethernet or even another protocol would be preferable to serial in the long run, but the serial is relatively simple to get to work and fills the requirements for this stage of the development.

The serial receiver hardware is as already noted connected to the spare spillover port in the fourth spawn unit. The serial receiver itself takes 14-byte instruction packets, with each byte being

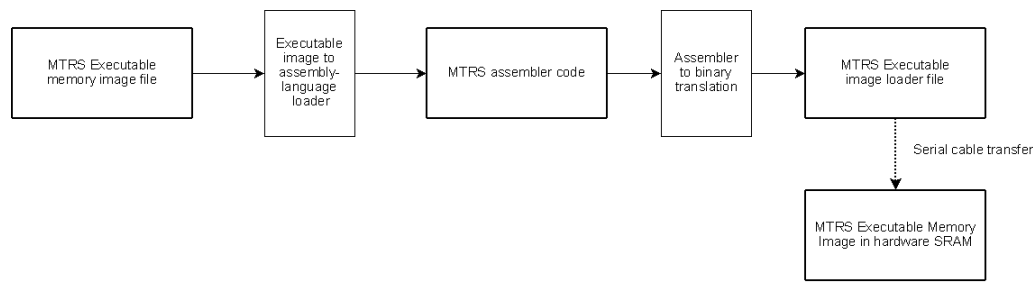


Figure 3.3: Schematic for loading an executable image into the FPGA SRAM.

transferred over RS-232 in turn, reformats them for internal processor use, and inserts them into this spillover port, wherefrom they subsequently get executed in the processor. The use of 14-byte data here means that the processor does not know the difference between an instruction just spawned from RAM or one inserted via this extra external means, meaning that any instruction can be sent and executed just as if it was loaded from memory. The serial cable communication implementation is coded together with the rest of the processor in Verilog and occupies space in the same FPGA as the processor implementation.

Any instruction thus encoded correctly and sent over the serial connection in a 14-byte format will be executed as soon as the MTRS processor can handle it. This most importantly applies to 'store' instructions, which can write to the block RAM in the FPGA and the memory mapped serial output. Several other instructions are of very limited usefulness when used over the serial cable as any instructions sent in this fashion will not have any allocated stack space and each instruction will be treated as completely independent, thus not sharing any information with preceding instructions transferred. Jump instructions are for example perfectly useless, as jumping to a point in RAM is of no use while transferring the program. Transferring a spawn instruction is on the other hand useful, for it can be used after the programming is complete to start execution.

Only acting on 14-byte data packets does however limit the technique and forces all registers to be filled in all instructions in a program that is transferred in this way. Furthermore, should there be a problem in transmission, there is no reset on the 14-byte count, and it is likely that the prototype processor implementation have to be completely reset to bring it back into order.

3.3.2 Software

The software for the program loader consisted of a particular piece of software, referred to as 'mem2prog', designed to convert an already assembled program into a program in assembly code that does nothing but write the assembled program into RAM. This means that a program that is expected to run on the processor first gets assembled like a normal program, and the binary image which is effect the MTRS executable code, is stored on the host system. This is then taken by the 'mem2prog' program and is redone into another assembler level program that only consists of

'store' instructions with the binary data of the assembled program as the data to store. This second program is then assembled into binary, and the binary instructions are sent over serial cable to the MTRS prototype processor. The hardware then inserts these instruction packets into the processor pipeline, and the processor dutifully executes them, thus writing the program that is intended to be executed into the FPGA block RAM. The software thus in effect wraps the desired program in MTRS code that is executed on the processor to load the program into RAM.

This writes the program to the RAM, but it does so through effectively one-shot instructions over the serial cable. To execute the work program itself from RAM, a spawn instruction needs to be sent over the serial cable as well. This spawn instruction instructs the processor to initiate a new thread using instructions from address 0x0002; and to do it in a non-continuing fashion so that the spawn instruction sent over serial is discarded. This is where the first instruction is located in the assembled instruction streams, as memory location 0x0000 is already used for a uniform zero value and 0x0001 is used for a global pointer.

3.3.3 Summary

The program loader transfers compiled programs to the MTRS prototype using serial communications, and by encapsulating the program into another MTRS program that gets executed. This approach was the simplest to implement while still retaining an efficient method and not using too much extra hardware. The method also means it is possible to test the processor without loading instruction into RAM, but rather execute code directly injected into the omega network. This was very beneficial during testing of the processor.

3.4 Compiler

The compiler is responsible for translating high-level language to low-level (assembly) language. The main reason for this is to increase programmer productivity; it is much easier for a programmer to write high-level code compared to assembler code. The compiler bridges the gap between high-level code and assembly code, just as the assembler bridges assembly code and the executable representation. Compilers are in general less specialised to a processor architecture than assemblers, but there is still an element of specialization as not all code is supported on all processors.

This section outlines the compilation process used for the MTRS implementation, starting with compiling C code to MIPS assembly and then translating MIPS assembly to MTRS assembly. The details of this process is presented and shown to produce viable MTRS code from C source code. The quality of the code thus produced is briefly examined and areas of improvement are discussed.

3.4.1 Motivation and Approach

Having created an assembler capable of assembling code from a text format assembler description the next step is to create a full compiler that can transform a higher-level language into a suitable assembler format, in particular into the prototype MTRS format. This may be seen as an extra unnecessary step having already proven that the MTRS prototype is correct by executing assembled code. The purpose here is to be able to execute already-written programs on the MTRS architecture, and also show that the microthreaded property of the processor can be abstracted by the compiler. This is important for a number of reasons, most importantly is that it shortens system implementation time, eases high-level program complexity and thus makes it cheaper to build a full computer system. Other reasons are for system adoption; it is easier to transition to a new computer system if all existing software works on it. These are business rather than technical considerations and as such are well worth adhering to. We need a way to translate high-level code into suitable MTRS assembler.

The compiler should be able to compile arbitrary high-level code, but for the purposes of this project, the example and test code used is not taken from an existing code base but created specifically for the purpose. It is considered sufficient if the compiler can compile a subset of a high-level language; the purpose is not to show a full featured compiler but rather that compilation can be performed correctly and to outline the process of how to do it. Improvements and modifications to the compilation process are expected in order to make a full featured compiler. As such, no source code for the compiler will be shown. Instead a set of steps is presented as a way of accomplishing the compilation. These steps were implemented as part of the project and the discussion revolve around the properties of the implementation, but even so the implementation is not final but intended as a prototype compiler.

3.4.2 Compiler structure

A relatively simple way of generating a compiler for a new architecture is to use an old compiler and write a new back end to it. In effect, this utilises the proven technology of the parent compiler as a base for the new; the assumption is that the new compiler back end will be relatively easy to implement and will still generate correct and efficient code. In particular, tested and verified high-level code optimisations can be used by the parent compiler to improve the code, saving implementation time and costs while still providing an optimizing compiler.

Following this reasoning, it was decided to use primarily GCC¹, the Gnu Compiler Collection, to compile code for the MTRS prototype. GCC would handle translation from high-level language into an assembly level representation, which would then be transformed into MTRS assembler and assembled into executable code. Further, MIPS assembler code was chosen as the intermediate

¹<http://gcc.gnu.org/>

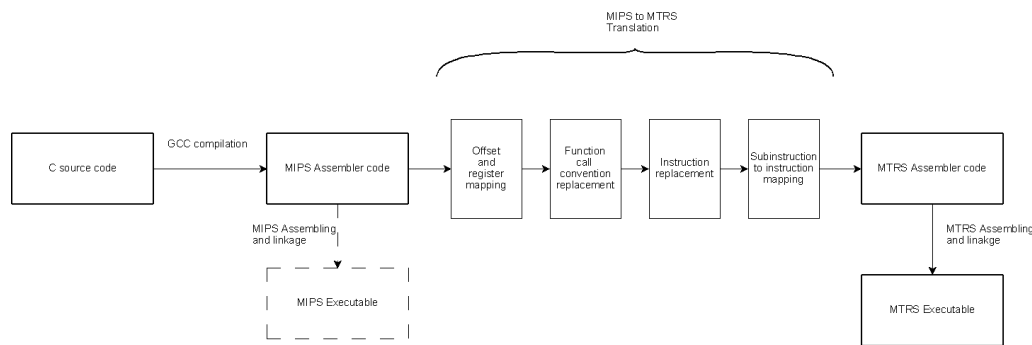


Figure 3.4: Schematic of the compilation process to produce MIPS executable code from C source.

step as it is relatively uncomplicated as instruction sets go, and would thus provide a good mapping to the limited assembler available in the MTRS prototype. GCC also has a broad range of code optimisations available which with this method can be used to improve the performance of MTRS.

This left transforming to MIPS to MTRS assembler as a problem to solve. This was accomplished by rewriting the MIPS code one step at a time, replacing MIPS instructions and paradigms with MTRS ones. Some of the steps are dependant on each other, but for others the order they are performed in is immaterial; the rewrites presented here is in the order that was finally used. In effect, this created a n-pass code transformer working over a set of instructions, at each step ensuring that the internal workings of the instructions remain the same.

The structure of the compiler is shown in figure 3.4.

Translating to Threaded Code

A large part of the processor concept is the microthreading concept. To take full advantage of this, the compiler should translate linear code into microthreaded code. This is a difficult problem to solve in general as constructs commonly used in linear code, such as global variables and pointer indirection, are hard to parallelise as accesses to these data areas have to be synchronised or might lead to incorrect execution. Due to constraints on this project limited effort have gone into making the compiler complete this requirement. Even so this does need to be addressed, and this section is intended to do so.

Listing 3.1 shows a linear C code example that computes two values through two functions, named `function1()` and `function2(char * text)`. These functions are assumed to be independent in that they do not share state, further they are assumed to be defined elsewhere and available at compile time. What they do is largely immaterial, it is assumed that they compute some int value from given parameters. In the case of `function1` it might be some value dependent on external state, in `function2` it might be some kind of hash mapping. As written, this code snippet is linear, in that `function1` is executed first and `function2` is executed afterwards. It is however obvious that these functions can be executed in parallel, if the compiler can extract the inherent parallelism.

```
1 int main(int argc , char * argv [])
2 {
3     int value1 = 0;
4     int value2 = 0;
5
6     value1 = function1 ();
7     value2 = function2 ("test");
8
9     // some other code here ...
10
11     printf ("%i %i\n" , value1 , value2);
12 }
```

Listing 3.1: Linear code example in C.

To take advantage of this the compiler should rewrite the function calls to read `function1 (int * ret)` and `function2 (int * ret, char * text)`. This would allow the functions to turn into microthreads with a cleanly defined return parameter, i.e. by spawning the functions as microthreads with this call signature they have a clear handle back to their parent microthread that can be used to pass the results back. This inserts a variable named `ret` into the functions; this variable is assigned the resultant value as the return from the function. In effect, the compiler needs to rewrite the functions themselves to replace any instances of `return value;` with `*ret = value; exit ();` Which returns the value to a variable in the address space of the calling microthread and then terminates the thread instead of just transferring control back to the calling function.

In addition to rewriting the callee functions in this way, the compiler needs to insert code into the parent microthread to synchronise the microthreads before using the results. Synchronisation does not need to happen until right before the data values are to be used, and further synchronisation does not need to suspend the parent thread. The reason for this is that MTRS is expected to have a large number of threads active at once, and having one or even several of them using busy-wait as synchronisation at once is not a problem. Furthermore, if there is a hardware memory manager, it might be able to detect the busy-wait pattern and suspend the thread until the variable it is waiting on has been assigned to.

What should happen is that the code should be rewritten to instead resemble the code in listing 3.2. The spawn function introduced is merely to show that the functions are to be spawned as microthreads, and this is likely not to be implemented as an actual function but rather a compiler transformation. As can be seen, none of these transformations have changed the strength of the code, and they can be applied to generic C code. Other program constructs may not be so simple to parallelise, and require further synchronisation, but this can again take the form of busy-waiting. The code listed in 3.2 is ready to have the function references turned into microthreads, which can then be scheduled concurrently into the MTRS processor, as performed by the proof-of-concept MTRS compiler. It should be noted that the outlined source to source transformation has not been

```
1 int main(int argc , char * argv [])
2 {
3     int value1 = 0;
4     int value2 = 0;
5
6     spawn( function1(&value1) );
7     spawn( function2(&value2 , "test") );
8
9     // some other code here ...
10
11    while( value1 == 0)
12    {}
13    while( value2 == 0)
14    {}
15    printf( "%i %i\n" , value1 , value2 );
16 }
```

Listing 3.2: Linear code example in C transformed for concurrent calls of function1() and function2().

included in the proof-of-concept compiler outlined, as the focus was on low-level code generation and processor hardware generation.

Compiling to MIPS

C source code was compiled to mips assembler code via GCC v3.3, running in a native MIPS environment. No optimisations were used for the example presented, in order to render the resultant code more legible for the subsequent examination of the translator. Optimisations could be used as seen fit but in the presented example this has the distinct possibility of removing clarity or introducing operations that are hard to emulate on the prototype MTRS processor.

The C routine chosen for this is a simple multiplier as shown in listing 3.3. A multiply routine is used as an example as the hardware lacks a multiply unit and this routine would be a necessary part of any software system developed on this processor. This function assumes, without explicit declaration, that an `int` is unsigned, as negative numbers are not handled; further, it assumes that the result will fit into an integer, as there is no checking for overflow. The function parameters for this simple multiply routine might look a bit unusual; using a pointer as a return value mechanism instead of the standard value passing is intended to simplify threading by allowing each function call to become a microthread (see sections 3.4.2, 2.1.2 and 1.1.2). The reason for using not-equal instead of greater-than on line 4 of listing 3.3 is because the underlying MTRS implementation lacks a compare-greater-than and compare-less-than function(see section 2.3.3 and A.1.3.2). With an improved compare functional unit this could be turned into a normal greater than zero operation; as it is the current processor implementation lacks this capability this peculiarity can be corrected by an improved MTRS implementation.

As can be seen, with the current MTRS implementation as outlined in chapter 2 it is not

```
1 void multiply(int * result , int x, int y)
2 {
3     int res = 0;
4     for (; x != 0; x--)
5     {
6         res = res + y;
7     }
8     *result = res;
9 }
```

Listing 3.3: Multiply routine as C code.

currently possible to use off the shelf C code and recompile for MTRS; it is expected that this situation can be improved. Most of the limitations are specific to the given implementation, and are not underlying faults in MTRS itself. Thus, if the compiler implementation and the processor implementation were to be further developed it is expected that the MTRS concept can be used with off the shelf C code.

Compiling the code in listing 3.3 with GCC and extracting the MIPS assembler code results in the code shown in the rather lengthy listing 3.4. This includes the stack/frame handling lead-in and lead out code, all of which is not relevant to the MTRS architecture, and which can be automatically stripped out. This code is replaced later with MTRS specific frame and stack handling code as well as implementation specific caller / callee semantics. The full MIPS listing is shown here for completeness. In addition to the machine-translated code shown, comments have been added to indicate what parts of the code refers to what part of the original C program.

Offset and Register Mapping

Having thus obtained a MIPS assembly listing from the C source code we need to modify the MIPS code into MTRS code. This is done through several steps, the first of which is as mentioned to cut out the lead in and lead out pieces of code that are MIPS specific, as they do not map at all to MTRS conventions. After this, we need to map the address offsets and register names to the MTRS convention; this is done as the MIPS and MTRS coding conventions in this area is not identical.

After cutting the lead in and lead outs the first step is to translate the hard coded address offsets into offsets that are more useful for MTRS. MIPS works on byte-addressable memory and 32-bit integers, whereas the current implementation of MTRS works on 16-bit addressable memory with 16-bit integers. Thus, the integer offsets into memory need to be altered. As this change could affect the layout of the frame, care was taken to have the new offsets relate to the old offsets by $n/4 - 4$. This maps offset 16 onto offset 0, offset 20 into offset 1, and so on, based on the observation that the offsets below 16 is used for MIPS-specific data which MTRS does not use. Figure 3.5 shows the assembler code after cutting out the lead in and out and applying this

```

1 multiply :
2 .LFB5:
3     .frame    $fp,48,$31
4     .mask    0x50000000,-8
5     .fmask   0x00000000,0
6     subu    $sp,$sp,48
7 .LCFI5:
8     sd     $fp,40($sp)      # stack and frame pointer handling
9 .LCFI6:
10    sd     $28,32($sp)
11 .LCFI7:
12    move   $fp,$sp
13 .LCFI8:
14    .set     noat           # lead-in section
15    lui    $1,%hi(%neg(%gp_rel(multiply)))
16    addiu  $1,$1,%lo(%neg(%gp_rel(multiply)))
17    daddu  $gp,$1,$25
18    .set     at
19    sw     $4,16($fp)      # Parameter passing and variable
20    sw     $5,20($fp)      # initialization. Everything above
21    sw     $6,24($fp)      # this can be removed as it is not
22    sw     $0,28($fp)      # relevant to MTRS.
23 .L6:
24    lw     $2,20($fp)      # Loop condition check.
25    bne   $2,$0,.L9
26    b     .L7
27 .L9:
28    lw     $3,28($fp)      # Main loop
29    lw     $2,24($fp)      # loads x and res
30    addu   $2,$3,$2        # res = res + y
31    sw     $2,28($fp)      # store res
32    lw     $2,20($fp)      # load x
33    addu   $2,$2,-1        # decrement x
34    sw     $2,20($fp)      # store x
35    b     .L6
36 .L7:
37    lw     $3,16($fp)      # store result and lead-out
38    lw     $2,28($fp)
39    sw     $2,0($3)
40    move   $sp,$fp
41    ld     $fp,40($sp)      # stack handling lead out
42    ld     $28,32($sp)
43    addu   $sp,$sp,48
44    j     $31
45 .LFE5:
46    .end     multiply

```

Listing 3.4: Multiply routine in MIPS4 assembler.

transformation. One point to note is that due to the store-word restriction on offset size (see section A.1.3.2), some extra handling is needed if there is more than three arguments to the function being compiled.

The ability to use integers as offsets for loads and stores were added late in development, when it was realised that MIPS used this semantic extensively, and while MTRS could work without it would require more registers than available to hold intermediate values. The instruction set was

then amended to include address offsets. At the same time, the instruction set was also changed to remove the notion of specific registers for instructions, a feature that while reducing instruction size also made the architecture very difficult to compile code for; as the compiler had to keep track of where results were stored. The instruction set rewrite is discussed further in section 2.2.1 and is also mentioned in A.1.1.

```

1 multiply :
2     sw     $4,0($fp)      # store values passed in
3     sw     $5,1($fp)      # registers
4     sw     $6,2($fp)
5     sw     $0,3($fp)      # initialize res
6 .L6:
7     lw     $2,1($fp)      # loop condition check
8     bne   $2,$0,.L9
9     b     .L7
10 .L9:
11     lw     $3,3($fp)      # loop body
12     lw     $2,2($fp)      #
13     addu  $2,$3,$2        # add y to res
14     sw     $2,3($fp)      # store res
15     lw     $2,1($fp)      # load, decrement, and store x
16     addu  $2,$2,-1
17     sw     $2,1($fp)
18     b     .L6
19 .L7:
20     lw     $3,0($fp)      # retrieve res, store to result
21     lw     $2,3($fp)
22     sw     $2,0($3)
23     j     $31

```

Listing 3.5: Multiply routine in MIPS4 assembler with modified address offsets and removed leadin and leadout.

The next step after rewriting the address offsets is to change the register names. Listing 3.6 shows the code after mapping all the registers to the appropriate names for MTRS. Register \$0 is used as a static zero value in MIPS, but this is not true in MTRS. All register numbers are therefore decremented by one; register \$0, where used, becomes an immediate '0' and which is later easily detected and the correct `.SET` can be inserted, as can be seen in listing 3.8. This occurs in this example on line 5 and line 8, where it is plainly visible that the \$0 has become '0'. Also notice that register names r4 and r5 are still present in this listing, on lines 3 and 4, even though these are not valid addresses in the current MTRS implementation. This is corrected in the next stage when adaptation to MTRS caller/callee semantics is inserted.

Function Call Convention Replacement

Having thus mapped the registers and address offsets, we need to insert the MTRS caller / callee semantics to ensure proper operation of the program. The MIPS code performing this function was already stripped out earlier, and we now need to replace it.

```

1 multiply :
2     sw    r3 ,0(r0)      # store values passed in
3     sw    r4 ,1(r0)      # registers
4     sw    r5 ,2(r0)
5     sw    0 ,3(r0)       # initialize res
6 .L6:
7     lw    r1 ,1(r0)      # loop condition check
8     bne   r1 ,0 ,.L9
9     b     .L7
10 .L9:
11     lw    r2 ,3(r0)      # loop body
12     lw    r1 ,2(r0)      #
13     addu  r1 ,r2 ,r1     # add y to res
14     sw    r1 ,3(r0)      # store res
15     lw    r1 ,1(r0)      # load x
16     addu  r1 ,r1 ,-1     # decrement x
17     sw    r1 ,1(r0)      # store x
18     b     .L6
19 .L7:
20     lw    r2 ,0(r0)      # store res in result
21     lw    r1 ,3(r0)
22     sw    r1 ,0(r2)
23     j     $31

```

Listing 3.6: Assembler multiply routine with the register references replaced.

Caller / callee semantics in the sample implementation of MTRS is simple: callers allocate a stack area of 8 16-bit values through a stack allocator. The stack allocator itself is located in hardware in the prototype implementation; this is not necessarily the case for a more complete implementation. On the allocated stack the values passed in are saved, starting at double byte location 0 and going up to 8. This is passed to the callee as register r0, and the callee can then retrieve data values; the callee also uses the stack to store its own local variables. Register r0 is therefore not to be touched by operations, just like in MIPS, but for a different reason. This synergizes with the MIPS semantics, which pass data through registers only after copying it to memory; the callee then saves the registers again. Thus we do not need to insert the copy data to memory instructions but just remove the saving to memory instructions in the callee. While passing data in registers is more efficient and useful when writing MTRS assembler, there are not enough available registers to do this in general. This may change if the implementation changes, as passing values in registers is a lot more efficient.

Listing 3.7 shows the multiply code after removing the superfluous save instructions have been removed. Note how the references to nonexistent registers have disappeared. In addition to the changes reflected in this example, the caller code is also modified to make these semantics work. The code size has again shrunk by removing MIPS-specific information in a callee, but due to extra instructions an example of a caller instead of a callee would have grown slightly in this stage.

```

1 multiply :
2     sw      0,3(r0)      # initialise res to 0
3 .L6:
4     lw      r1,1(r0)     # loop condition check
5     bne    r1,0,.L9
6     b      .L7
7 .L9:
8     lw      r2,3(r0)     # loop body
9     lw      r1,2(r0)     # load res and y
10    addu   r1,r2,r1      # res = res + y
11    sw      r1,3(r0)     # store res
12    lw      r1,1(r0)     # load x
13    addu   r1,r1,-1      # x = x - 1
14    sw      r1,1(r0)     # store x
15    b      .L6
16 .L7:
17    lw      r2,0(r0)     # load addr(result)
18    lw      r1,3(r0)     # load res
19    sw      r1,0(r2)     # store res in addr(result)
20    j      $31

```

Listing 3.7: Assembler multiply routine modified for MTRS callee conventions.

Instruction Replacement

With the function lead-in and lead-out in place we now need to replace the MIPS instructions with MTRS ones. While it would be possible to make the assembler understand the MIPS keywords, this is less than ideal as it opens up for confusion and incorrect behaviour due to slightly divergent effects of the instructions.

```

1 multiply :
2 .SET r1,0x0000
3     store   r1,3(r0)     # initialize res to 0
4 .L6:
5     load    r1,1(r0)     # loop condition check
6     bne    r1,0,.L9
7     b      .L7
8 .L9:
9     load    r2,3(r0)     # load y & res
10    load    r1,2(r0)     # load res
11    add     r1,r2,r1      # res = y + res
12    store   r1,3(r0)     # store res
13    load    r1,1(r0)     # load x
14 .SET r2,0xffff
15    add     r1,r1,r2      # increment x by 2^16-1
16    store   r1,1(r0)     # store x
17    b      .L6
18 .L7:
19    load    r2,0(r0)     # store res in result
20    load    r1,3(r0)     # load res
21    store   r1,0(r2)     # store res in addr(result)
22    j      $31

```

Listing 3.8: Assembler multiply routine with lw, sw and addu replaced with the appropriate instructions. MTRS highlighting.

The first instructions to get replaced are 'lw', 'sw' and 'addu', for load, store and adds respectively. In MIPS, these instructions work on 32-bit values, but as the MTRS implementation work on only 16-bit values; as both register and instruction sizes are the same in both cases, this distinction is ignored in this case. Listing 3.8 shows the result of completely replacing these instructions in the MIPS code; in addition, the highlighting has changed to MTRS code to more clearly illustrate the changes. Of particular interest in this code transformation is the insertion of a `.SET` on line 2 and 14. On line 2, this is providing the 0 value for the store on line 3, which initialized our local variable, as alluded to earlier. On line 14, it provides the -1 needed for the decrement of the index parameter. The underlying MTRS implementation does not support negative numbers and is treating all numbers as unsigned, but by adding `0xFFFF`, causing an addition overflow, the effect becomes the same. Line 14 also shows how add-immediate syntax is emulated on MTRS even though it is not supported in hardware; the effect of a `.SET` followed by an add to the same register has the same effect. This relies on this stage detecting free registers to use for immediate values, but even with the very limited registers available this can usually be accomplished.

The next to last step in transforming MIPS assembler syntax is to replace the jump instructions present. This is an area where MTRS and MIPS diverge a lot, and where care must be taken to ensure correct translation. Listing 3.9 shows the example code after this has been performed. In addition to this transformation, comments have been added to show what parts of the code corresponds to the C code in Listing 3.3.

This transformation first of all replaces the `'j $31'` sequence with a `'CALL end'` sequence, which serves no other purpose than to terminate the current thread. This is performed as a subroutine call is transformed into a microthread, and the microthread is have at this point finished executing. In addition, thread termination instructions should be inserted just above this; such instructions have not been included in the listing. Thread cleanup code amounts to handing the used stack back to the stack manager, be it hardware or software, and to make sure no spawned child threads are still active.

The other transformations carried out in this step are the rewrites of the branch instructions with `ceq` and `cneq` equivalents, and also inserting the branch addresses through `.SET` immediates. There is nothing complicated about this except mapping to the correct conditional test and inserting it. The only other transformation that has been done is to insert all labels as `.CALL` labels so that the assembler knows to start new instructions at these points.

Subinstruction Mapping

With the instructions translated to their MTRS equivalents, the instructions need to be allocated to MTRS instruction blocks. This is a problem related to instruction scheduling in conventional processors and compilers, and is in general difficult to solve. A very simple algorithm for inserting

```

1 .CALL multiply
2 .SET r1,0x0000
3     store    r1,3(r0)
4 .CALL .L6
5     load    r1,1(r0)
6 .SET r2,0x0000
7 .SET r3, .L9
8     cneq   r1,r2           # loop condition check
9     jump   r3             # branch to loop body
10 .SET r2, .L7
11    jump   r2,r1,r2       # branch to cleanup & terminate
12 .CALL .L9
13    load   r2,3(r0)       # load y and res
14    load   r1,2(r0)
15    add    r1,r2,r1       # res = res + y
16    store  r1,3(r0)       # store res
17    load   r1,1(r0)
18 .SET r2,0xffff
19    add    r1,r1,r2       # add x and 2^16-1
20    store  r1,1(r0)
21 .SET r2, .L6
22    jump   r2,r1,r2       # jump to loop condition check
23 .CALL .L7
24    load   r2,0(r0)       # cleanup code starts here
25    load   r1,3(r0)
26    store  r1,0(r2)
27 .CALL end

```

Listing 3.9: Assembler multiply routine with 'j', 'b' and 'bne' instructions replaced.

the breaks is used in this example, but it is expected that this can be performed with better results.

After having replaced all branches and jumps in the MIPS code the listing is all MTRS code. It will not yet compile correctly as it lacks the complete instruction limiters and adds entries. This is similar to omitting the next instruction pointer in a superscalar architecture, as the instructions generated from this would have no reference of where to go next if they do not have an explicit jump instruction. The last step therefore inserts these markers and completes the transformation to valid MTRS code. Listing 3.10 shows the result of this final transformation.

As can be seen, this transformation introduces extra information into the program relating to how MIPS instructions are packed into MTRS instructions. The **addr** instructions are added to indicate which is the next instruction to execute, and is inserted into each instruction so that the logical straight down behaviour is followed. This can arguably be termed the most complex transformation that is taking place, as the instruction blocks should be delimited in an optimal way. No attempt to do so was done, instead all breaks were inserted on a when needed basis. this has led to some strange placements of the block delimiters in that there are more then the necessary number of instructions with only one subinstruction. It can be seen that it is quite possible to make a more optimal placement of instruction block delimiters, but doing so in an automated fashion would require a more complicated algorithm or algorithms, and thus is subject for further work.

Despite this slight non-optimality, the code shown can now be processed by the assembler

```

1 .CALL multiply
2 .SET r1,0x0000
3     store    r1,3(r0)      # initialization of res
4     addr     .L6
5 .CALL .L6                # load x into r1
6     load     r1,1(r0)
7     addr     Lmultiply8
8 .CALL Lmultiply8        # loop condition check
9 .SET r2,0x0000
10 .SET r3, .L9
11     cneq    r1,r2 # .L9
12     jump    r3
13     addr    Lmultiply9
14 .CALL Lmultiply9        # continued branch condition check
15 .SET r2, .L7
16     jump    r2,r1,r2
17     addr    .L9
18 .CALL .L9                # loop body
19     load    r2,3(r0)
20     load    r1,2(r0)
21     add     r1,r2,r1
22     store   r1,3(r0)
23     addr    Lmultiply10
24 .CALL Lmultiply10        # loop body, continued
25     load    r1,1(r0)
26     addr    Lmultiply11
27 .CALL Lmultiply11        # loop body, further continued
28 .SET r2,0xffff
29     add     r1,r1,r2
30     store   r1,1(r0)
31     addr    Lmultiply12
32 .CALL Lmultiply12        # loop body, jump to loop check
33 .SET r2, .L6
34     jump    r2,r1,r2
35     addr    .L7
36 .CALL .L7                # finishing code starts here
37     load    r2,0(r0)      # storing res in addr(result)
38     load    r1,3(r0)
39     store   r1,0(r2)
40     addr    end
41 .CALL end                # finished

```

Listing 3.10: Assembler multiply routine converted to MTRS.

previously outlined and generate executable code that can be expected to run on the prototype MTRS processor.

3.4.3 Testing

The compiler was tested in stages as it was created, with test programs being fed through each stage in turn and inspected at the end of that stage to ensure that the correct transformation was carried out. Each compiler stage was then adjusted until it generated the correct output for the test cases, and work was then started on the next stage in the chain. This method of construction of the compiler ensured that each stage was correct before beginning the next, and led to the entire

compiler being correct by constructing each intermediate stage and testing it to be correct. The main test program used was primarily the multiply example already expanded on in Listings 3.3 through 3.10, but also including context around this example so that caller / callee semantics was tested. This test program had defined transformations for each stage in the compiler chain, and thus was useful for detecting these construction tests.

A further source of potential problems were eliminated by using standard libraries wherever possible, for example for regular expression matching and string manipulation. The use of standard libraries ensure that problems with these functions have a significantly lower chance of occurring as these libraries are assumed to have been tested many times before and can therefore be regarded as problem free. There is still a chance that there are issues that stem from the use of these libraries, but the chance is significantly lower than if these routines were written specifically for this project. In the same vein was also assumed that the use of GCC to compile from C to MIPS code was essentially bug free, as the very same version of GCC is shown to be working by compiling system software for native MIPS computer systems. GCC is further assumed to already have been tested exhaustively, and therefore no tests were executed for this particular stage in the compiler chain.

The most comprehensive test is to execute code from the compiler on the MTRS prototype processor itself. This test ensures that both the processor and the compiler are to some degree correct and more importantly that they are both correct enough to produce viable code and to execute it.

3.4.4 Code Quality

Optimal code for the MTRS prototype processor would have every bit in the generated executable RAM image be meaningful to the program execution, and furthermore to not incorporate redundant instructions. In general, optimal code would execute the minimum number of instructions, encoded in a minimum number of bits, needed to execute the task presented. The instruction set used in the prototype processor is not optimal to this degree, as it contains bits that may be unused or redundant, for example the zero arguments to the **nop** instruction or repeated arguments to the **add** instruction.

The code generated by the current compiler implementation is not optimal in this sense. There are several features of the code that takes more space and bits than strictly necessary, even given the existing instruction set. First of all, the GCC translation to MIPS assembler code does not currently use any compiler optimisations. Transformations at this level have the power to reorder and optimise the code on the non-local level and is a very powerful tool in optimisation. This is as already stated not done for reasons of code clarity. Using these high-level optimisations would be a first step in optimising code for the MTRS platform.

Apart from the lack of these high-level transformations, there are problems with the generated

code that needs addressing in developing the MTRS concept further. None of these issues invalidate the MTRS concept or idea, but they are important for obtaining the full execution speed and efficiency possible from the MTRS concept.

Firstly, there is a problem with packing instructions into instruction blocks that will need to be addressed. Packing less than four instructions in a block means that there will be more instruction block loads for a given number of subinstructions, which increases the memory bandwidth pressure. In the prototype implementation this is not a major problem due to the memory set-up, but it will be important going forwards due to limited memory bandwidth. It is also the case that instruction blocks are fixed-length and at least 48 bits long; any **nop** instructions inserted will cause loads of unused bits which again wastes memory bandwidth. For these reasons it is important that a better algorithm for packing instructions into instruction blocks are developed.

Certain structures such as a while-zero loop can be written in MTRS assembler as a single instruction block and is used frequently in code. The compiler currently might try and split this over several instruction blocks, depending on the exact code employed, which is non-optimal. A single instruction block is easier to keep cached and reused, cutting down on memory bandwidth once again. It would be necessary have the compiler know about common structures that can be implemented in particularly efficient ways and have it map code to these ways when possible. There is no implementation of this in the current compiler.

There is some redundant memory traffic generated by the dearth of registers in the prototype processor itself. This is not a compiler problem, but it becomes a problem when values need to be spilled and restored repeatedly due to this lack of registers. The compiler does not currently have a perfect model of spilling / restoring registers, which would be necessary for using the few registers available in the best way possible. This is an issue that can be solved by either increasing the number of registers in the processor or improving the compiler.

The code quality generated by the compiler is for these reasons not perfect, but can be considered a proof of concept compiler together with the proof of concept processor. If the processor concept is to move forward, it is expected that compiler would require significant improvements as well, in particular in the field of generating efficient code.

3.4.5 Summary

Compiling code for MTRS is complex due to the microthreaded model of the processor. Automatically parallelising code that is being compiled is a problem that is still regarded as hard, in that it is computationally demanding and usually does not produce as good results as rewriting the code to be explicitly parallel does. Even so, the MTRS compiler has to try and parallelise code as the benefit of the processor is expected to be in the parallel workloads it enables. The parallelising problem for MTRS is not as important as for other architectures, as it is also assumed that there

are enough tasks to perform in the MTRS system that parallelising code is the largest hinderance to performance.

The MTRS compiler compiles code from a higher-level language into MTRS assembly code, that can then be processed by the assembler to generate executable code. The compiler works by compiling high-level C code to the MIPS instruction format using the GCC compiler without optimisations, which is a standard and proven compiler. Thus the step of translating C to MIPS code is not expected to have any errors in it and can be relied on.

The MIPS assembler code is then translated to MTRS assembler code in a number of steps that each change the code in a small way. There are four major steps in this process, each comprised of several small steps. In total they transform every aspect of the MIPS assembler code to MTRS assembler code, including inserting microthreading constructs on function calls. The resultant code implements function calls as microthreads in the MTRS processor, and uses busy-waiting as a synchronisation mechanism. This is the limit of parallel extrapolation carried out in the compiler, something that could be improved with further development.

Other improvements to the compiler would be necessary before it can be considered as something other than proof of concept. It would have to optimise the code generated better, to reduce the memory bandwidth problem, and to decrease the space taken by compiled programs. The need for these improvements does not detract from the MTRS concept in itself, but would only make it more viable, as they are improvements on the efficiency of the generated code. Even lacking these improvements, the compiler is functional and can compile code that will execute on the prototype MTRS processor, thus proving that the MTRS concept is a viable way of building processors and also fulfilling the goals of the project.

4 Conclusion

The concept of a reduced state microthreaded processor has been investigated in detail, and it has been shown that this concept is viable both theoretically and practically. The aim of the project was to investigate the viability of this design concept in both theory and practice. This aim has resulted in a prototype processor design and an associated software toolchain to compile software for the prototype processor. The project aims can therefore be considered more than fulfilled.

Microthreaded reduced state processors share similarities with VLIW, Superscalar, SMT, and Microthreaded processor architectures, but cannot be classified directly as either. At best it can be considered as combination of VLIW and SMT technologies, but even this analogy is weak and fails to grasp many of the important parts of the MTRS concept such as the emphasis on reduced state. Even so, features from these architectural concepts have been integrated into the MTRS prototype where appropriate.

Using the MTRS design ideas as a base and choosing structures that fit the model, such as an omega network and pipelined functional units, a prototype processor has been constructed and extensively tested. This prototype has been shown to work in simulations as well as in a FPGA fabric. The constructed prototype processor shows that it is possible to build processors using the reduced state and microthreaded design paradigm without suffering obvious impairment of any property that would reduce the functionality of the processor. The prototype processor executes code correctly, as checked through both simulations and practical testing.

The associated compiler and assembler for the processor prototype have also been discussed in detail and have been shown to generate executable code for the processor. The construction of the compiler received less effort than the processor construction, but even so the compiler and assembler are essentially complete and capable of compiling code for execution on the prototype processor. The assembler and to some extent the compiler have been tested by compiling code for the processor itself and executing the compiled code on the FPGA implementation of the prototype processor, thus showing that the compiled code is correct and by extension that the assembler and compiler are correct.

A brief look at the completed system indicates that it is suited for applications which benefit from parallel processing. The completed MTRS system would also benefit full computer systems that use a large number of parallel tasks, such as transaction servers and database servers, as the processor is designed with microthreads as an inherent property. Despite being intended for these massively parallel applications, the MTRS prototype is relatively simple in hardware terms; it is sizeable, but not particularly large compared to other contemporary processors, and most of its structure is repeated many times.

These concluding sections discuss the design and implementation of the project from the per-

spective of further work and what can be done differently. It is acknowledged throughout the project that the current implementations of both the compiler and processor are prototypes, and as such are expected to be the subject of further refinement. The basic ideas of the MTRS processor remains the same, and it is also likely that the broad structure of the processor and compiler will remain the same, but many of the details can be changed and tweaked in other ways than what was done for the prototype.

4.1 Processor

The processor design is essentially complete to the point of being a functional processor that can execute compiled code. Further work on the processor itself would largely be targeted at introducing more features into the processor, in particular features that have been omitted or neglected. There is a number of areas in which the processor can be improved to either execute code more efficiently or execute code that previously used a software workaround.

If the processor is developed further, the conditional functional unit can be extended to include support for greater than and not greater than operations, as these are very common but are not yet supported natively in the prototype processor, but can be emulated in software if necessary, a solution that is not optimal.

There are a number of operations that could be added to the processor to alleviate omissions or make it more efficient; most notably there is a need to support operations such as subtraction, bitwise and, bitwise or, and bit shifts. These operations are commonly used, but are currently not supported natively. It is debatable whether these operations should all be included in the same functional unit or if they should be split up into two or more functional units; something that would need to be investigated before these operations are incorporated. It is also possible to include other operations, such as integer multiplication and integer division, but the inclusion of these operations would need to be evaluated and justified.

The processor is designed to operate on 16-bit values as a tradeoff between numerical precision and instruction block length. If the processor is being extended, this trade off might have to change; using 32-bit data is a common feature of current microprocessors. Changing this would involve changing the size of instruction blocks and the widths of the paths in the omega network, and will lead directly to a higher gate count of the processor, something that would necessitate the trade off.

There should arguably be support for different lengths of operations; for example use of the registers as 8-bit characters for string manipulation operations is expected to be very common. Using the registers as vectors of smaller data values is another feature that could be considered; for example, with eight 32-bit registers one could design additional functional units to handle the

registers as two four by four matrices with 8-bit values in each matrix position instead, allowing for native matrix operations. Combining this with functional units that address the same eight 32-bit registers as 8-bit values could be a useful combination. The use of synergistic functional units like these would need to be researched and justified, in particular with a view towards gate count as instructions would likely be extremely long with the associated register data.

None of these methods of extending the processor is however useful without first creating the memory management unit to work together with the processor. The memory management unit is likely to be on the same order of complexity as the processor prototype, and is expected to handle process control blocks, synchronisation, caches, and the interface to main memory. This is a tall order, but the system is not complete without this unit. As has been shown with the prototype processor, it is possible to make a MTRS processor without a MMU, but for any practical implementation the MMU is required. The design of the memory management unit has from the start been considered to be outside the scope of the current project, and aside from the parts of the processor that required a subset of the functionality of the MMU, this was adhered to. The first and most obvious extension of the project is therefore to design and implement the MMU for the MTRS processor.

One final improvement to the processor would be to implement it in real silicon instead of a FPGA, after having thoroughly tested the design in simulations or in an FPGA fabric. Such an improvement would potentially allow the processor to reach clock speeds on the order of several hundred MHz, and allow a serious evaluation of the processor performance and capabilities, a subject which has not been touched upon in this project.

4.2 Compiler and Assembler

The assembler and compiler for the MTRS prototype are complete in that they can produce code for the prototype processor implementation, but even so they lack several important properties. These deficiencies could be the goal for further work, in particular with a view to make the compiler less of a prototype and take it further towards an full strength reliable compiler.

The first issue that would need to be addressed with the current compiler is its dependance on MIPS assembler code as an intermediate step. It would be more useful if this step could be omitted, and the compiler could translate directly from intermediate representation into MTRS assembler. This would make the compiler much more efficient and save having to strip off the irrelevant MIPS assembler sections and insert the corresponding MTRS sections. To accomplish this one would have to in effect write a new back end to an existing compiler, which despite being difficult could conceivably be accomplished as an extension to the project.

One area where the current compiler is extremely deficient is in optimising the code generated;

there is currently no code conversions done to optimise for any variable. This would be a very useful addition to the compiler toolchain, as it would allow the compiler to produce code that performs better on the MTRS platform. Potential optimisations include the classical computer science optimisations such as constant propagation, subexpression evaluation and propagation, and loop transformations, as well as MTRS specific optimisations, which could be for example determine where to split into a new microthread. MTRS specific optimisations would additionally need to be researched and their effectiveness determined.

Packing instructions into instruction blocks is a non-trivial problem, but the proof of concept compiler does not spend a large amount of time on the problem. This could be significantly improved, leading to more efficient code being produced by the compiler. The compiler should not insert a **nop** into the instruction stream unless it absolutely have to, as such instructions have to be loaded from RAM but does no useful work.

The Instruction Set Architecture of the processor is nonoptimal in several respects, and if the processor is being improved at the same time as the compiler there may be a good reason to change some of the choices made in the ISA. The compiler and particularly the assembler would then need to be updated and improved to handle the new instruction formats and instructions introduced.

4.3 System

One area where the development of the processor is so far lacking is in basic performance evaluation. Due to the nature of the project goals there is not yet a comprehensive performance evaluation of the MTRS concept, as the project was focused on the ability to construct a MTRS processor. Evaluating the performance of the MTRS concept is important to determine if the design concept is viable in practice. The project has shown that it is possible to build the MTRS processor, but has not addressed questions regarding the real-world efficiency of the MTRS processor.

Such an evaluation would have to be performed using various different MTRS processor designs, with varying interconnect network sizes and different functional unit combinations, to establish that the measured performance is typical for the processor family as a whole. Each of the processors would need a compiler tweaked to its particular instruction set or alternatively have a common instruction set. Either of these approaches would potentially also need to be evaluated so as to firmly establish the importance of the compiler in the MTRS concept.

These tests cannot be performed until the MTRS concept has been developed further, as the current implementation is not mature enough to be used for this purpose. Further development of the MTRS concept and in particular the MTRS implementation is needed, possibly along one or more of the lines given in the above sections, before comparative performance of the MTRS with regards to hardware resources and power consumption can be accurately determined.

Acknowledgments

Many thanks to Björn Franke, for had he not agreed to supervise this project, it would have been on an entirely different topic.

Thanks also to Archie Howitt, for unwavering support in practical matters, in particular in regards to sourcing a replacement FPGA board halfway through the project.

References

- [1] O. Almer, “MEng Project Report Part 1: Design and Simulation of a Reduced State Microthreaded Processor Architecture.” June 2007.
- [2] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3rd ed., 2003.
- [3] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, “Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor,” in *ISCA*, pp. 191–202, 1996.
- [4] J. A. Fisher, “Very long instruction word architectures and the ELI-512,” in *ISCA*, pp. 140–150, 1983.
- [5] P. Kongetira, K. Aingaran, and K. Olukotun, “Niagara: A 32-way multithreaded SPARC processor,” *IEEE Micro*, vol. 25, no. 2, pp. 21–29, 2005.
- [6] R. M. Russell, “The CRAY-1 computer system,” *Commun. ACM*, vol. 21, no. 1, pp. 63–72, 1978.
- [7] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpfen, M. Frank, S. Amarasinghe, and A. Agarwal, “Evaluation of the RAW microprocessor: An exposed-wire-delay architecture for ILP and streams,” in *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, (Washington, DC, USA), p. 2, IEEE Computer Society, 2004.
- [8] J. R. Gurd, C. C. Kirkham, and I. Watson, “The Manchester prototype dataflow computer,” *Commun. ACM*, vol. 28, no. 1, pp. 34–52, 1985.
- [9] K. Bousias, N. Hasasneh, and C. Jesshope, “Instruction Level Parallelism through Microthreading—A Scalable Approach to Chip Multiprocessors,” *The Computer Journal*, vol. 49, no. 2, pp. 211–233, 2006.

A.1 Appendix: Instruction Reference

The MTRS assembler instruction reference outlines what assembler instructions are available in the prototype MTRS implementation. The prototype does not have a lot of features, and the instruction reference is thus relatively short.

A.1.1 Development

The instruction set presented here is currently in its second iteration. There were severe problems in the first iteration that necessitated a redesign.

These first of these problems were generally a lack of immediate operations coupled with a severe lack of registers in the instruction set. While the lack of registers was already decided on and difficult to change, adding immediate operands to certain functions was a relatively easier redesign. It can be seen that the 'load' and 'store' instructions now take a 3 and 2 bit additive offset respectively. This greatly enhances the compiler/assemblers ability to translate MIPS code to MTRS, and is also expected to facilitate developing a stand-alone compiler for MTRS.

The second problem that required addressing was that some operations, notably add operations, had an implicit third argument it used to store the results in. This argument was dependant on where in the instruction block the add was present; in other words, the semantics of the add operation changed with how it was placed in a block. It was found that this was very difficult to compile code for, as the assembler would often schedule an instruction block with only one instruction in it, namely the add, and the rest filled out with 'nop' instructions. This was highly inefficient, and stemmed from the semantics of the add instruction. It was found that this problem went away entirely with the use of a three-argument add instruction with no implicit arguments, and the add instruction was duly changed to this format.

A.1.2 General instruction format

Instructions in the MTRS architecture are generally referred to as instruction blocks, which in turn contain subinstructions. The instruction blocks are issued as a full unit to the processor, and are retired when there are no more subinstructions left in the instruction block, or when a suitable jump subinstruction is encountered. The general format of the instruction block is four 9-bit instruction specifiers, containing both subinstruction code and data area specifiers, one 10-bit address specifier, 2 reserved bits, and four 16-bit data areas, which may or may not be preloaded with data, for a total maximum instruction block size of 112 bits / 14 bytes. This sequence is stored in memory as lowest 16-bits first, most significant bit first, and continuously in 7 16-bit chunks. Figure A.1.1 shows the generic load instruction block process graphically.

The four 16-bit data areas are generally referred to as registers, despite not being registers in the classical sense. The MTRS architecture has no register file, in keeping with the Reduced State goal. The data fields are in general 4 extra 16-bit latch banks which allows data to traverse the processor together with the associated instruction blocks. The data areas are available only to the current instruction stream, and have no global presence in the processor. These data areas are named r0 through r3, in order, and are used as such in the assembly code. Due to there only being four of these data areas available at any time, the pressure on registers is very high. It is expected that any non-trivial program on the reference MTRS architecture utilise multithreading (as data areas are instruction-stream, i.e. thread, local, more data fields can be used through multithreading) and a high percentage of loads and stores.

The subinstructions are executed in the order they are listed in the instruction block. No effort is made towards out-of order execution. Similarly, no effort is made towards branch prediction. These features are possible in the MTRS architecture, but are not present in the prototype MTRS design.

Instruction blocks follow each other through the `addr` instruction. This is not an instruction in itself, but merely fills in the 10 address bits allocated in the instruction. When an instruction

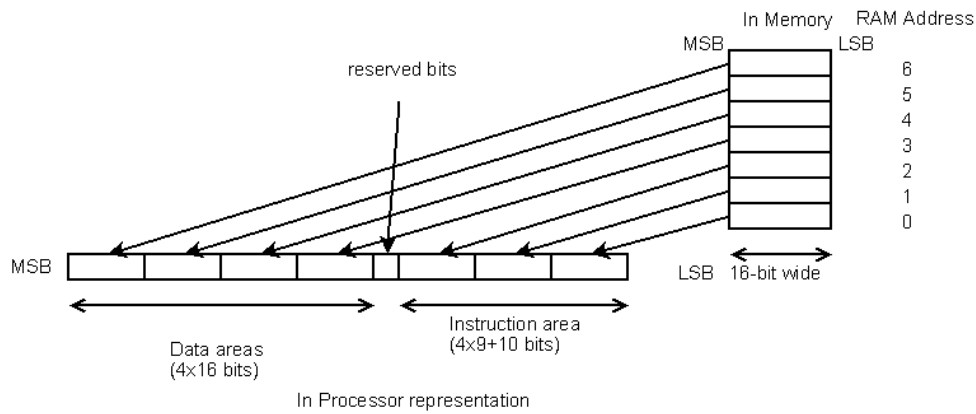


Figure A.1.1: Relation between in-memory and in-processor representations. Arrows denote where 16-bit values go. This illustration assumes all .SET variables are used.

has executed all its subinstructions, an automatic instruction to jump to this address is inserted by the MTRS processor, and the instruction located at this address is then loaded. This saves having a jump instruction as the final instruction in every instruction block, which would impact code efficiency.

A.1.3 Reference Assembler

The MTRS reference assembler assembles straight-line assembly code to straight-line binary code. It does not do any transformations on the code. The herein presented assembler directives are translated straight to their binary equivalents.

An instruction block, in heavily optimised code, may be as small as just the 4 sub-instruction areas, provided the data fields are all inherited from the previous instruction block, as allowed by the jump instruction. In this way, it is possible to have variable size instructions by omitting the data areas from the instruction stream. This is mostly useful to keep total code size down by not having long runs of 0 values in the binary code; this is especially important on the MTRS prototype platform as it has a relative dearth of RAM.

The assembler instructions are listed herein with the following fields:

Assembler instruction: The textual representation interpreted by the assembler.

Arguments: what arguments the assembler instruction takes

Binary equivalent: The binary number the instruction gets translated to

Argument equivalents: the binary number(s) the arguments gets translated to

These are intended to make it easier to understand the generated binary code. Note that while there are more than one of several functional units (spawn and add each have four in the prototype implementation) the exact functional unit a subinstruction gets executed in is of no consequence, and is generally dependant on what subinstruction it came from. As each functional unit of each type is equivalent, it also does not matter which one is used. Which unit is used is in theory deterministic, but with more than one thread executing at once it is effectively random, which means that the load placed on a particular functional unit type is balanced across the number of functional units of that type available. This is a feature of the interconnect network chosen for the MTRS prototype.

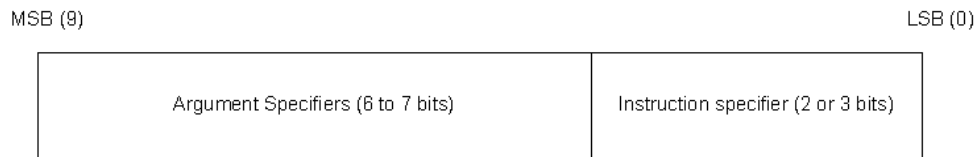


Figure A.1.2: Generic subinstruction layout in memory and in flight through the MTRS processor.

A.1.3.1 Non-Instructional Operatives

.CALL

Indicates to the assembler that a new instruction block should start here. the assembler syntax is:

`.CALL label`

This indicates to the assembler that any values of *label* in the file should be replaced with an address to this newly started instruction. This is particularly useful for assigning to `.SET` of a data value used by a jump. A label is comprised of a string of letters and numbers that does not start with '0x' or ''.

.SET

Indicates to the assembler that the next data field of the instruction should be set. The assembler syntax is:

`.SET n,value`

Where *n* is the register to set (this must conform to any register that is set by the previous jump). The reference compiler makes no sanity checks on the link between a previous jump and the registers specified for `.SET`. The *value* can be a 16-bit number in hexadecimal format, with a mandatory '0x' prefix; a two character ASCII string enclosed in citation marks (" "); or a label (See `.CALL`).

An instruction can have up to 4 `.SET`s defined for it. These are entered into the data fields in the order of the `.SET` statements. It is recommended that to avoid confusion the register part of the `.SET` is always sequential, as follows:

`.CALL label`

`.SET r0,value`

`.SET r1,value`

`.SET r2,value`

`.SET r3,value`

It is possible to match the 'footprint' of these registers against the one used in jumps, and thus guarantee that the data values end up in the correct places, but this is not used in the reference assembler.

.VAR

Should be used to define data values that cannot be fitted into a `.SET`; and that therefore needs load / stores to be used against it. the assembler syntax is:

`.VAR label,data`

Where *data* is the data that should be represented. This can be maximum 16 bits long, and follow the same guidelines as for data in `.SET`. The *label* is a label associated with this data, so that it can be referenced in a `.SET`.

A.1.3.2 Sub-Instructions

Subinstructions are the constituent parts of an instruction block. There are 4 subinstructions in an instruction block. Subinstructions are generally on the format shown in figure A.1.2, with varying bit field lengths. There is a total of 6 available subinstructions in the reference design. A fully usable design is expected to have many more possible subinstructions; however, the 6 chosen for this design are sufficient for a processor to be Turing-equivalent and thus enough to show the functionality of the MTRS concept.

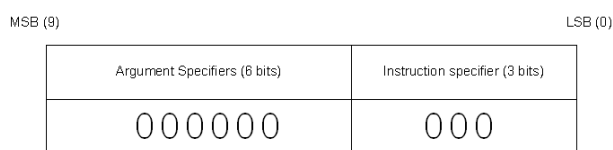
Nop

Assembler instruction: nop

Arguments: none

Binary Equivalent: 0x0, 3 bits long

Argument Equivalents: 0x0, 6 bits long



Nop is the classical do nothing instruction. It delays the instruction block by 1 cycle and then inserts the instruction into the datapath again.

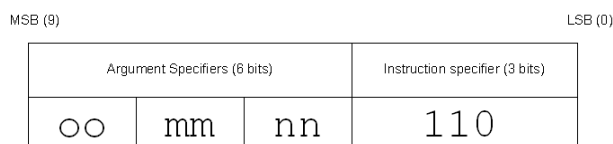
Store

Assembler instruction: store

Arguments: $rm, n(ro)$

Binary Equivalent: 0x6, 3 bits long

Argument Equivalents: $0x_m, 0x_n, 0x_o$, 2 bits long each



Store takes the address from register m , adds the offset o and stores the values from register n there. This operation takes 2 clock cycles to release the instruction back into the datapath, and 2 instructions to assert the write lines to RAM. Note how the offset argument is only 2 bits long, and so can't handle offsets larger than 3.

Important memory locations for the reference design is location 0, (0x0000), which is always set to 0, and cannot be written to. Also, location 0xF000, which is the location of the 16-bit area that is to be written to serial output at a rate of 115200 baud.

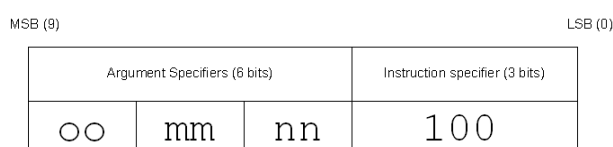
Conditional

Assembler instruction: ceq, cneq

Arguments: rm, rn

Binary Equivalent: 0x4, 3 bits long

Argument Equivalents: $0x_n, 0x_m$ 2 bits long each



Cond is the conditional operator. It advances execution to the next subinstruction, if the registers pointed to by the arguments are either equal (ceq) nor not equal (cneq), otherwise it forwards execution two subinstructions. The field marked 'oo' in the binary representation determines if this is a ceq or a cneq; if it is '00' it is a ceq, if it is '01' it is cneq. The values 10 and 11 are reserved.

Using a conditional instruction as the next to last instruction in an instruction block is valid, as skipping the last instruction leads to loading the next instruction block from the set address. Using it as the last instruction in an instruction block does not do anything.

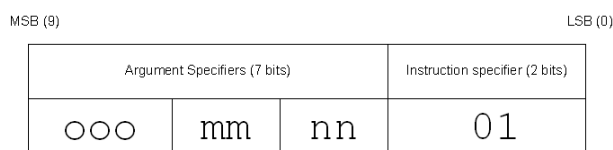
Load

Assembler instruction: load

Arguments: *rm*, *rn*, *offset*

Binary Equivalent(s): 0x1, 2 bits long

Argument Equivalents: 0xn, 0xm, 2 bits long each



Load loads a value from RAM. The address to load from is taken to be in the register *m* adds the 3-bit offset *ooo*, and stores the retrieved result in register *n*. This subinstruction has a delay of three cycles before it reinserts the instruction block into the datapath.

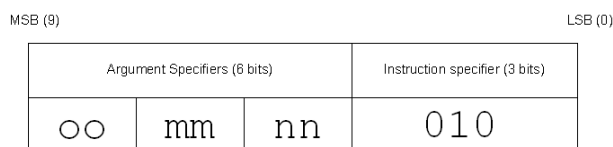
Add

Assembler instruction: add

Arguments: *rm*, *rn*, *ro*

Binary Equivalent(s): 0x2, 3 bits long

Argument Equivalents: 0xn, 0xm, 0xo 2 bits long each



The add instruction adds the two registers *m* and *n*, and stores the result in register *o*. The addition is done as 16-bit unsigned 2-s complement addition, and does not check for overflow in any way.

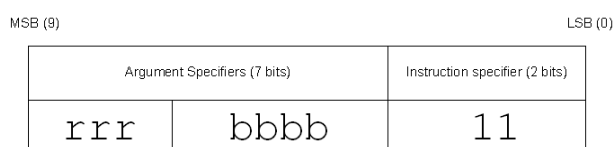
Spawn and Jump

Assembler instruction: jump

Arguments: *rm*, plus any of *rn*, *ro*, *rp*, *rq*

Binary Equivalent(s): 0x3, 2 bits long

Argument Equivalents: bit pattern matching what registers were given



The jump instruction loads a new instruction block from RAM. The address of the block is given by the first register argument.

The argument to jump is a bit field describing which registers to duplicate into the new instruction block, and which to populate from memory. Any registers mentioned in the subinstruction syntax are to be replaced with registers loaded from RAM together with the instruction block.

A.1.3.3 Example

A conditional instruction block would, in its shortest form, have the following format in assembly language:

```
.CALL if-statement  
.SET r1,label to jump to if equal  
.SET r2,label to jump to if not equal  
cond rn,rm  
spawn register list  
spawn register list
```

the .CALL locates the instruction block. Next, the .SETs set up the addresses to jump to. These are picked up automatically by the jump that issues this instruction block. The cond then compares the two values needed (it would make sense that these values are r0 and r3, as these are inherited from the previous instruction block; this is however not a requirement). depending on if they are equal or not, the first or the second jump are encountered next. It is possible for both to be encountered, if the comparison is equal and the address in r1 is uneven; this issues the new instruction block AND continues the current one. This block would have to be called by a 'jump r1,r2' to ensure that the .SETs gets picked up. This is a clear example of how the footprint of the jump and the .SETs must match.