$\mathcal{RRISC}$ Architecture Report

Zak Smith Jeffry Lucman Jeremy Petsinger
John Liu Mostafa Arifin

December 12, 1997

Contents

Chapter 1

Executive Summary

During the Fall semester of 1996, the $\mathcal{RRISC}$ team designed, debugged, and built a "non–trivial computer with an original instruction set," as specified in the project requirements for ECE554 at the University of Wisconsin — Madison.

After specification and design, the computer was implemented in hardware using Xilinx XC4000 series FPGAs, with support circuitry on the WICEPS board supplied for the course.

The main feature of our computer was that it was super–scalar; it could execute up to 3 instructions simultaneously. To conserve limited resources, we limited our instruction word length to 13 bits, and our data word length to 8 bits. The computer had 8k words of instruction memory and 16k words of data memory.

The system was composed of 4 main modules. The dispatch unit handled scheduling of instructions for execution. The AIO, ALU, and MEM units were each 3–stage pipelines mostly independent of each other which executed subsets of the instruction set. These execution units operated in parallel.

The instruction set was based on standard reduced instruction set ideas (RISC), with the exception of a few special instructions to make memory address calculations easier, which would have been difficult since our data and instruction word lengths differed.

The project was a success: once in hardware and debugged, the system ran at 4.09 MHz, a period of 209ns.

Although ECE554 has one of the highest workloads of any undergraduate ECE class, it is one of the most worthwhile and most satisfying. We all gained existential joy when our programs ran on the computer we had designed and built.

Chapter 2

Introduction

2.1 Purpose

The mission of UW–Madison's ECE554 students is to

Design a non–trivial computer with an original instruction set.

This document describes the computer built by the $\mathcal{RRISC}$ team which consisted of: team leader Zak Smith, Jeffry Lucman, Jeremy Petsinger, John Liu, and Mostafa Arifin.

2.2 Background

All of the students in the class have taken ECE552, the introductory computer architecture class, in which they designed a simple CPU in a sterile simulation–only environment. In ECE554, however, the teams also physically build the systems, taking into account real–world limitations of the equipment available and engineering decisions about the tradeoffs.

In addition, the group must learn *how to work successfully in a group.* This often is overlooked, but it can mean the difference between success and failure.

2.3 Organization

Following this introduction is the "Design Spec" section, which describes general architecture and specification, and then the "Implementation Details" section explains each block's design. Finally, the "Results & Conclusions" chapter discusses limitations and choices we would change if we had to do it over again. The appendices contain the Instruction Set Architecture specification and a matrix describing each member's contributions.

2.4 Additional Reading

Readers not familiar with basic comptuer architecture issues should refer to *Computer Organization & Design: The Hardware / Software Interface*, by Patterson and Hennessy.

Technical specifications of the hardware provided for class use are available in the ECE 554 class notes.

Chapter 3

Design Spec

3.1 Overall Architecture

We have developed a superscalar architecture, and have named it $\mathcal{RRISC}$: the Radically Reduced Implementation of a Superscalar Computer.

Superscalar means there are multiple pipelines to which instructions are issued simultaneously. There are two ALU pipes and one memory access pipe. Data hazards must be detected by the compiler (or assembly programmer) and `NOP`'s must be inserted into the instruction stream. There are similar restrictions for branch instructions.

3.2 Word Size

The $\mathcal{RRISC}$ uses 8–bit data words and has 13–bit instruction words. (These "odd" numbers were chosen because the FPGA's have limited routing and pin resources, and increasing either number would make our overall architecture untractable.)

3.3 Memories

The $\mathcal{RRISC}$ has disjoint instruction and data memories, which are addressed using 14–bit and 13–bit addresses, respectively. There are 8192 words of data memory, and 16384 words of instruction memory in two 8192 word parallel banks. The two instruction memory banks each use the upper 13 bits of the instruction address. The last bit is used to discriminate between the two instructions inside the processor.

3.4 General Purpose Registers

The $\mathcal{RRISC}$ has 8 general purpose registers `$0` ... `$7`. Reads from `$0` always return `0x00` and writes to `$0` have no effect. The other registers, `$1` ... `$7` are normal registers.

3.5 Special Registers

The $\mathcal{RRISC}$ has several special purpose registers. The first is the obvious `PC`, the program counter. The programmer modifies this value with the jump and branch instructions. Since the registers are only 8 bits, we needed several special registers to hold 14 and 13–bit addresses. The `SP` is a stack pointer which supports several push and pop type instructions. The `AR` acts as a general address register, which is used for link addresses, absolute jumps, and load and store word instructions. The `PC`, `AR`, and `SP` are 14–bit registers.

Branching is done based on the `SET` bit, which is a single bit register holding the result of the last test instruction executed. All instructions which use the

ALU, with the exception of AND and OR, set the CY carryout bit, which can be used to do arithmetic beyond 8 bits.

3.6 Scheduling

NOP's in the instruction stream act as barriers between sets of dependent instructions. All instructions before a NOP will be issued before the NOP is issued. NOP's are issued in order to all pipes simultaneously — that is, when a NOP is the next instruction in the dispatch stream, all three pipelines will received a NOP signal simultaneously. The easiest programming paradigm, given this structure, is to write code in clusters of orthogonal instructions, and separate those clusters each by a single NOP. The only restriction for jumps and branches is that the instruction immediately following a branch must not be a jump or a branch.

The guidelines for avoiding data hazards are as follows: dependent register instructions must be seperated by one NOP, or enough other instructions to keep the pipes busy for 2 cycles. After AR has been popped from the stack, one NOP or other instruction to keep the memory pipe busy is required before another instruction which requires AR, like `ret` or `jf`.

3.7 Physical Constraints

The system was implemented in an array of 6 Xilinx XC4000 FPGAs which reside on the WICEPS board supplied by the instructors for the course. Each FPGA has $14^2 = 192$ configurable logic blocks. Each CLB contains 2 flip–flops, and the capability for arbitrary logic functions based on two 4x1 bit RAMs and one 3x1 RAM fed by the two four 4x1's with an additional input.

Each FPGA chip has 107 usable pins, which contrains how the design can be distributed between the chips. This also limits the word length and the width of busses which must pass between chips. Each FPGA also has limited internal routing capability, so care must be taken to minimize the number of internal signals.

Chapter 4

Implementation Details

This chapter provides an overview of the system on a functional–block level. Details of individual functional blocks are described in the next chapter.

4.1 Major Functional Blocks

The system was developed in five main functional blocks: dispatch, master ALU pipe, slave ALU pipe, memory pipe, and register file. Each functional unit resided in a single FPGA. This division was chosen mainly in order to not run out of pin resources.

4.1.1 Dispatch

The dispatch unit is responsible for issuing instructions to each of the execution pipes. The dispatch reads two instructions in parallel from instruction memory, and provides the instruction and instruction–enable signals to each execution pipe by the next clock edge. Instruction stalls can only occur in the dispatch unit, and they occur when either of the instructions are destined for the same execution pipe or a jump or branch is propagating down the memory pipe.

4.1.2 Master ALU Pipe

The master ALU executes ALU and IO instructions and keeps track of the ALU special registers `SET` and `CY`. The master ALU FPGA also contains the TIS.[1]

4.1.3 Slave ALU Pipe

The slave ALU executes only ALU instructions, and communicates changes in the `SET` and `CY` registers to the master ALU.

4.1.4 Memory Pipe

The memory pipe executes all other instructions, which includes those instructions with modify the address register `AR`, the stack pointer `SP`, and the program counter `PC`.

4.1.5 Register File

The register file is an eight by 8–bit register file implemented in D flip–flops. The register file is clocked on the negative edge of the system clock in order to allow read–after–write. Reads from register `$0` always return `0x00` and writes to `$0` have no effect.

[1] Terminal Interface System developed as miniproject 2

4.2 Dataflow

Instruction words move from the instruction memory to the dispatch. The dispatch issues instructions to the three execution pipelines, which when take between one and three cycles to complete execution. Instructions do not move between execution units as they are being executed.

4.3 Control

There are several block–level control signals which are used to manage state and status changes between the units. When the dispatch unit stalls, it asserts *NOFETCH* which causes the memory unit not to write the incremented `PC` on the next clock edge. If the dispatch unit stalls because a jump or branch instruction has been issued, it does not continue until it receives *PCC* from the memory pipe, which means that the jump or branch instruction has finished executing and the data from instruction memory is now the new, correct instruction. The memory pipe will assert *CONT* if a branch was issued, and it has executed but the branch has not been taken. This will cause the dispatch to issue the "delayed" instruction when the branch was located on the even address. The instruction immediately following a branch on an odd address cannot be a jump or a branch because it violates the stipulation that only one `PC`–modifying instruction will be in the dispatch at any one time.

The dispatch sends a "*NOP*" flag to each execution pipeline which invalidates the instruction word going to that pipe. When *NOP* is active for a certain pipe, the instruction word is disregarded, and no action is taken.

4.4 Collision Detection

Although data dependencies must be handled by an intelligent assembler, compiler, or programmer,[2] the hardware will raise the *TRAP* signal and halt the `PC` when two or more instructions try to write to the same register at once. Checks are made for `SET`, `CY`, and `$1` . . . `$7`.

The *TRAP* signal is wired to an LED on the WICEPS board to alert the programmer to his error. We have found this also useful for halting a program during debug.

[2]This proved more difficult than one would think.

Chapter 5

Results & Conclusions

5.1 Limitations of $\mathcal{RRISC}$

The largest limitation is the fact that we use only 13-bit instructions. This limits us in many ways. First, our jump addresses are only eight bits, which means that for any jumps farther than +/- 128 instructions, which is most of the time, we must first reference a pointer table, load the `AR`, then jump far, which is very time consuming.

The second major limitation is the fact that we only have 8 registers. In the ideal case, our superscalar machine executes 2 instructions per clock cycle, which means for normal arithmetic operations, we run out of registers very quickly.

The third limitation is largely a function of time. With more time or more people in the group, we could have built a smart compiler that converted high level code into optimized $\mathcal{RRISC}$ code. This would mean it issues do–nothing instructions to some of the pipes while keeping the other pipes running. This would avoid the use of `nops` to solve potential data dependencies.

The empirical maximum clock frequency was found to be 4.9 MHz, 204ns. We think that this limit was either inside the Dispatch unit, which had long FF to FF delays, or that is was due to the register file write, which had to happen during the PHI1 clock. The delay for a register write is approximately F/F to PADS + worst PADS to F/F:

$$20 + 63 = 83ns$$

This period should be within active high clock period because the data will be written on the negative PHI1 clock edge, and becomes active after the positive edge of PHI1. This yields a theoretical maximum clock period of $100/25*83ns = 332ns$, which is a frequency of 3.01 MHz.

5.2 Epilogue

After completion of the $\mathcal{RRISC}$ microprocessor, the team will disband and return to their old lifestyles. John will begin his job at intel in January, citing the fact that he likes working a lot as his main job selection criteria. Mostafa will begin graduate school in January, and is considering retaking 554 because he likes Mentor so much. Jeffry will continue his undergraduate studies.

Zak will graduate in May, work the summer in crazy California, and come back for more punishment as a graduate student in the Fall of 1997. Asked about his future plans, Jeremy responded "I'm kind of tired, I think I might go to bed."

Appendix A

Instruction Set

A.1 Operation Types and Instruction Word Formats

There are six defined instruction types, to facilitate the opcode and immediate fields. All instructions start with a 4–bit opcode.

Type	Layout	Used for
R-Type	$\overbrace{\text{Opcode}}^{4}\overbrace{\texttt{rd}}^{3}\overbrace{\texttt{r1}}^{3}\overbrace{\texttt{r2}}^{3}$	`add sub and or addc not`
M-Type	$\overbrace{\text{Opcode}}^{4}\overbrace{\texttt{rd}}^{3}\overbrace{\text{Imm}}^{6}$	`lrl lw sw larl`
S-Type	$\overbrace{\text{Opcode}}^{4}\overbrace{\text{Op Ext}}^{3}\overbrace{\texttt{r1}}^{3}\overbrace{\texttt{r2}}^{3}$	`slt seq`
O-Type	$\overbrace{\text{Opcode}}^{4}\overbrace{\text{Op Ext}}^{3}\overbrace{\texttt{r1}}^{3}\overbrace{\text{Imm}}^{3}$	`iotr iora iot ior laru sll srl sra pushr popr`
N-Type	$\overbrace{\text{Opcode}}^{4}\overbrace{\text{Op Ext}}^{3}\overbrace{\text{Imm}}^{6}$	`pushl pushh popl poph sptar ret nop artsp`
J-Type	$\overbrace{\text{Opcode}}^{4}\overbrace{\text{Op Ext}}^{1}\overbrace{\text{Imm}}^{8}$	`bs bns jn jf jaln jalf`

A.2 Notational Conventions

For purposes of operation specification, the following notation has been defined:

Notation	Meaning
`RD`	Register addressed by `rd`
`R1`	Register addressed by `r1`
`R2`	Register addressed by `r2`
$A \leftarrow B$	Value in B is copied into A
$A \leftrightarrow B$	B is exchanged with A
$A \vert B$	A bitwise catenated with B
$A \Leftrightarrow B$	A occurs if and only if B occurs
$A \Vert B$	A occurs in parallel with B
$\overleftarrow{\texttt{s}}\ A$	Sign–extend A
0_n	Pad with 0's
$\texttt{RD}_{7_n}$	Pad with `RD`'s MSB
$--A$	Pre–decrement A
$A++$	Post–increment A
$\texttt{M}[A]$	Data Memory location A

Note that the special registers are denoted `CY`, `PC`, `SP`, and `AR`. Memory is denoted with `M`[*location*], and registers specified in the instruction are denoted `RD`, `R1`, and `R2`.

A.3 Common ALU Operations

Opcode	Type	Pipe	Description
add	R	ALU	$\mathtt{RD} \leftarrow \mathtt{R1} + \mathtt{R2}$
sub	R	ALU	$\mathtt{RD} \leftarrow \mathtt{R1} - \mathtt{R2}$
and	R	ALU	$\mathtt{RD} \leftarrow \mathtt{R1} \wedge \mathtt{R2}$
or	R	ALU	$\mathtt{RD} \leftarrow \mathtt{R1} \vee \mathtt{R2}$
addc	R	ALU	$\mathtt{RD} \leftarrow \mathtt{R1} + \mathtt{R2} + \mathtt{CY}$
not	R	ALU	$\mathtt{RD} \leftarrow \neg\mathtt{R1}$
sll	O	ALU	$\mathtt{RD} \leftarrow (\mathtt{RD} \ll \mathtt{Imm}_3)\vert 0_n$
srl	O	ALU	$\mathtt{RD} \leftarrow 0_n\vert(\mathtt{RD} \gg \mathtt{Imm}_3)$
sra	O	ALU	$\mathtt{RD} \leftarrow \mathtt{RD}_{7_n}\vert(\mathtt{RD} \gg \mathtt{Imm}_3)$

A.4 Memory Access Operations

Opcode	Type	Pipe	Description
lw	M	MEM	$\mathtt{RD} \leftarrow \mathtt{M}[\mathtt{AR} + \mathtt{Imm}_6]$
sw	M	MEM	$\mathtt{M}[\mathtt{AR} + \mathtt{Imm}_6] \leftarrow \mathtt{RD}$
pushr	O	MEM	$\mathtt{M}[\mathtt{SP}++] \leftarrow \mathtt{R1}$
popr	O	MEM	$\mathtt{R1} \leftarrow \mathtt{M}[--\mathtt{SP}]$
pushl	N	MEM	$\mathtt{M}[\mathtt{SP}++] \leftarrow \mathtt{AR}_{7:0}$
pushh	N	MEM	$\mathtt{M}[\mathtt{SP}++] \leftarrow 0_2\vert\mathtt{AR}_{13:8}$
popl	N	MEM	$\mathtt{AR}_{7:0} \leftarrow \mathtt{M}[--\mathtt{SP}]$
poph	N	MEM	$\mathtt{AR}_{13:8} \leftarrow \mathtt{M}[--\mathtt{SP}]$

A.5 I/O Operations

Opcode	Type	Pipe	Description
iotr	O	AIO	$\mathtt{RD} \leftarrow 0_7\vert TBR$
iora	O	AIO	$\mathtt{RD} \leftarrow 0_7\vert RDA$
iot	O	AIO	$XMIT \leftarrow \mathtt{RD}$
ior	O	AIO	$\mathtt{RD} \leftarrow RECV$

A.6 Special Register Operations

Opcode	Type	Pipe	Description
laru	O	MEM	$\mathtt{AR}_{13:8} \leftarrow \mathtt{RD}_{5:0}$
larl	M	MEM	$\mathtt{AR}_{7:0} \leftarrow \mathtt{RD}$
sptar	N	MEM	$\mathtt{AR} \leftarrow \mathtt{SP}$
artsp	N	MEM	$\mathtt{SP} \leftarrow \mathtt{AR}$

A.7 Jump and Branch Operations

Opcode	Type	Pipe	Description
`slt`	S	ALU	$\mathtt{SET} \leftarrow (\mathtt{R1} < \mathtt{R2})$
`seq`	S	ALU	$\mathtt{SET} \leftarrow (\mathtt{R1} == \mathtt{R2})$
`jn`	J	MEM	$\mathtt{PC} \leftarrow \mathtt{PC}+ \overleftarrow{\mathtt{s}}\ \mathtt{Imm}_8$
`jf`	J	MEM	$\mathtt{PC} \leftarrow \mathtt{AR}$
`jaln`	J	MEM	$\mathtt{AR} \leftarrow \mathtt{PC} \ \Vert\ \mathtt{PC} \leftarrow \mathtt{PC}+ \overleftarrow{\mathtt{s}}\ \mathtt{Imm}_8$
`jalf`	J	MEM	$\mathtt{AR} \leftrightarrow \mathtt{PC}$
`bs`	J	MEM	$(\mathtt{PC} \leftarrow \mathtt{PC}+ \overleftarrow{\mathtt{s}}\ \mathtt{Imm}_8) \Leftrightarrow \mathtt{SET}$
`bns`	J	MEM	$(\mathtt{PC} \leftarrow \mathtt{PC}+ \overleftarrow{\mathtt{s}}\ \mathtt{Imm}_8) \Leftrightarrow \neg\mathtt{SET}$
`ret`	N	MEM	$\mathtt{PC} \leftarrow \mathtt{AR}$

A.8 Miscellaneous Operations

Opcode	Type	Pipe	Description
`lrl`	M	MEM	$\mathtt{RD}_{5:0} \leftarrow \mathtt{Imm}_6$
`lru`	O	ALU	$\mathtt{RD}_{7:6} \leftarrow \mathtt{Imm}_2$
`nop`	N		*bubble*

Appendix B

Examples

We designed the instruction set with simplicity and utility in mind. Here are a few examples of how to utilize the instruction set to do common tasks.

B.1 Function Call

This illustrates how a function called by `JALN` or `JALF` would save the return address and any registers onto the stack.

```
function:
pushl          ; push AR(7:0) onto stack
pushh          ; push AR(13:8) onto stack
pushr $1       ; save any registers we clobber
pushr $2       ;
pushr $3       ;
               ;
               ; function code goes here
               ;
               ;
popr $3        ; restore the registers we saved
popr $2        ;
popr $1        ;
poph           ; restore AR(13:8)
popl           ; restore AR(7:0)
ret            ; return to AR
```

B.2 Arithmetic Example

```
lrl $1 0x06    ; $1 = 06H
```

```
nop
add $2 $1 $0   ; $2 = 06H
sub $3 $0 $1   ; $3 = FAH
nop
not $4 $1      ; $4 = F9H
sll $2 0x01    ; $2 = 0CH
nop
and $5 $1 $2   ; $5 = 04H
or $6 $1 $2    ; $6 = 0EH
srl $4 0x01    ; $4 = 7CH
sra $2 0x03    ; $2 = 01H
```

Appendix C

Individual Contributions

C.1 Zachary Smith

As group leader Zak was involved in nearly every phase of $\mathcal{RRISC}$ developoment. He was heavily involved in developing the architecture and instrucion set including rough block descriptions of all major system components and he defined the functions of each pipeline stage. He designed, entered, and debugged the dispatch unit, and wrote the Life program for the demonstration.

C.2 Mostafa Arifin

Mostafa was primarily responsible for the design and testing of the two ALU pipes, and interfacing these units with the TIS. He was also heavily involved in Register File development, and assisted Mr. Lucman in many hardware and wirewrapping related activites. He also wrote the menu screen for the demonstration.

C.3 John Liu

John was involved in the original definition of the architecture and instruction set. He also wrote the original MAD, and the subsequent 10 revisions. MAD made code generation fast and easy.

C.4 Jeffry Lucman

Jeffry contributed in many areas of $\mathcal{RRISC}$ development including Register File design and testing, hardware tests on all FPGAs and memory modules, and several smaller components found in the ALU and Memory Pipes. Perhaps his most appreciated accomplishment was the error–free wirewrapping of the entire board. Jeffry also wrote the Sort program in optimized DELA code for the demonstration.

C.5 Jeremy Petsinger

Jeremy was heavily involved in the development of the instruction set and definition of the $\mathcal{RRISC}$ Architecture. Additionally, Jeremy designed and tested the Memory Pipe, and wrote matrix routines in DELA code, which demonstrated realistic throughputs on $\mathcal{RRISC}$ optimized code.

C.6 Accomplishment Matrix

Task	Zak	Jeremy	Mostafa	Jeffrey	John
Gimmick	20%	20%	20%	20%	20%
Organization Block	35%	35%	10%	10%	10%
Instruction Set	34%	34%	7%	7%	18%
Final Organization	100%	-	-	-	-
Pipe Functions	50%	50%	-	-	-
POO	100%	-	-	-	-
Dispatch	-	-	-	-	-
Design	100%	-	-	-	-
Entry	100%	-	-	-	-
Simulation	100%	-	-	-	-
Regfile	-	-	-	-	-
Design	-	-	40%	60%	-
Entry	-	-	30%	70%	-
Simulation	-	-	50%	50%	-
Memory Interface	-	-	-	100%	-
AIO Pipe					
Design	-	-	70%	-	30%
Entry	-	-	95%	5%	-
Simulation	-	-	100%	-	-
ALU Pipe					
Design	-	-	100%	-	-
Entry	-	-	100%	-	-
Simulation	-	-	100%	-	-
Memory Pipe					
Design	-	100%	-	-	-
Entry	-	95%	-	5%	-
Simulation	-	100%	-	-	-
System Simulation	22%	22%	20%	21%	15%
HardWare Debug	20%	20%	20%	20%	20%
Software Tools	90%	-	-	-	10%
Software Development	27%	34%	12%	27%	-
Dela	-	-	-	100%	-
FPGA Pin Test	-	-	40%	60%	-
Memory Test	-	-	-	100%	-
Path Delays	-	-	-	100%	-
Compute Stats	-	50%	-	50%	-
WireWrap	-	-	30%	70%	-
MAD	-	-	-	-	100%