$\mathcal{RRISC}$ Architecture Report [1] [2]

Zak Smith Jeffry Lucman Jeremy Petsinger
John Liu Mostafa Arifin

December 21, 1996

[1]We would like to thank the TAs Gebre Gessesse and Zhenyu (Jerry) Liu and Professor Kime for their help during the course of the semester.

[2]This is the condensed version of the report and as such does not contain the schematics or figures.

Abstract

The mission of UW–Madison's ECE 554 students is to *Design a non–trivial computer with an original instruction set.* This document describes the superscalar architecture developed by the $\mathcal{RRISC}$ team and how it was implemented in Xilinx XC4000 FPGAs using the WICEPS prototype board.

Contents

Chapter 1

Instruction Set Architecture

1.1 Overview

This chapter describes the instruction set architecture (ISA) of the $\mathcal{RRISC}$ system developed during Fall 1996.[1] The $\mathcal{RRISC}$ is a super–scalar architecture: 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. The specific numbers of `NOP`'s required for each instruction will be determined by the hardware implementation, and are not detailed in this chapter.

1.2 Architecture

1.2.1 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.)

1.2.2 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.

1.2.3 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.

1.2.4 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

[1] $\mathcal{RRISC}$ is an acronym for "Radically Reduced Implementation of a Superscalar Computer"

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.

1.2.5 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`.

1.3 Instruction Set

1.3.1 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`

1.3.2 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
$A\|B$	A bitwise catenated with B
$A \Leftrightarrow B$	A occurs if and only if B occurs
$A\|\|B$	A occurs in parallel with B
$\overleftarrow{\mathtt{s}}\ A$	Sign–extend A
0_n	Pad with 0's
$\mathtt{RD}_{7_n}$	Pad with RD's MSB
$--A$	Pre–decrement A
$A++$	Post–increment A
$\mathtt{M}[A]$	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.

1.3.3 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	$\mathtt{RD} \leftarrow (\mathtt{RD} \ll \mathtt{Imm}_3)\|0_n$
`srl`	O	ALU	$\mathtt{RD} \leftarrow 0_n\|(\mathtt{RD} \gg \mathtt{Imm}_3)$
`sra`	O	ALU	$\mathtt{RD} \leftarrow \mathtt{RD}_{7_n}\|(\mathtt{RD} \gg \mathtt{Imm}_3)$

1.3.4 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	$\mathtt{M[SP++]} \leftarrow 0_2\vert\mathtt{AR}_{13:8}$
`popl`	N	MEM	$\mathtt{AR}_{7:0} \leftarrow \mathtt{M[--SP]}$
`poph`	N	MEM	$\mathtt{AR}_{13:8} \leftarrow \mathtt{M[--SP]}$

1.3.5 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	$\mathtt{RD} \leftarrow RECV$

1.3.6 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	$\mathtt{SP} \leftarrow \mathtt{AR}$

1.3.7 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	$\mathtt{PC} \leftarrow \mathtt{AR}$

1.3.8 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*

1.4 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.

1.4.1 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
```

1.4.2 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
```

Chapter 2

Machine Architecture

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

2.1 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.

2.1.1 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 the instructions are destined for the same execution pipe or a jump or branch is propagating down the memory pipe.

2.1.2 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]

2.1.3 Slave ALU Pipe

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

2.1.4 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`.

2.1.5 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

2.2 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.

2.3 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.

2.4 Collision Detection

Although data dependencies must be handled by an intelligent assembler, compiler, or programmer, 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.

Chapter 3

Memory Subsystem

The RRISC uses seperate Instruction and Data Memory modules.

3.1 Instruction Memory

The Instruction Memory consists of 16K of 13–bit instructions using two 8K X 16–bit memories where bits 15–13 are don't care. The first memory contains even memory address instructions and the second memory contains odd memory address instructions. In order to load the instruction memory, a memory separator program divides a continuous instruction stream into even and odd addresses. On each memory read, the upper 13 bits of the Program Counter, $\texttt{PC}_{13:1}$, are provided as the memory address, from the Memory Pipe, and two instructions are issued to the Dispatch Unit in parallel.

There is no facility to write to instruction memory, so WR_BAR is tied high, and the chip output is always enabled.

3.2 Data Memory

The Data Memory consists of 8K 8–bit data words and interacts solely with the Memory Pipe. Data Memory uses one 8K X 16–bit memory module where the bits 15–8 are don't cares. The memory address is determined by $\texttt{AR}_{12:0}$ or $\texttt{SP}_{12:0}$ from the Memory Pipe. The 8–bit data, on a bidirectional data bus from the Memory Pipe is read on WR_Enable_bar high and written on WR_Enable_bar low. In order to ensure a stable data address, PHI2_BAR clock signal is used to access memory.

Chapter 4

Dispatch Subsystem Design

4.1 Overview

4.1.1 Introduction

The dispatch unit is responsible for the flow of instructions from instruction memory to the three execution units.

4.1.2 Registers

The dispatch contains two registers for the instruction words coming from instruction memory, and an "extra" instruction register which is used to hold the left over instruction in the case of a stall. The two instructions have been named M1 and M0, for the odd and even memory locations, and the "extra" buffer stage has been called B0. The dispatch also contains a "PC" register, which holds the upper 13 bits of the PC valid for the instructions coming from memory. The lower bit is held in its own 1–bit register controlled by logic to give the correct even or odd value when the instruction is issued.

4.1.3 Stalls

Stalls occur in two cases: the first is when there will be more than one instruction left, which means that at least two instructions are persisting in the dispatch for the next clock. In this case, there is not enough room for the two new instructions from memory, so the dispatch stalls until at least one of the remaining instructions clears. The other case in which a stall occurs is when a PC–modifying instruction is issued. If this happens, the "WAITING" flag, implemented in a J/K flip–flop, is set. It is not reset until the memory pipe reports that the PC–modifying instruction has cleared and the new instructions from memory are valid.

4.1.4 Instruction Types

There are three main instruction classes: pure ALU, named simply ALU; ALU/IO instructions, named AIO; and memory pipe operations, named MEM. For the non–overlapping instruction sets (ALU/AIO vs. MEM), the information must travel from B0 to M1. Overlapping set information (AIO vs. ALU) must travel from M1 to B0. The logic block used for each buffer stage control is the same: the multiple blocks are daisy–chained together, with the endpoints appropriately set to logic 0.

4.2 Fundamental State

The logic for the dispatch is built upon a base of fundamental state signals. These fundamental signals are generated for each buffer stage. In the table i is the stage we are at, M1, M0, or B0.

Name	Description
PCM	PC–modifying instruction
NOP	*i* is a NOP
ALU	*i* is ALU or AIO op.
TIS	*i* is an IO op.
B	*i* Blocks: *NOP* or *PCM*
rB	Blocking instruction at *i* or before
MEM	*i* is a MEM pipe op.
iTIS	*i* is TIS and not blocked
fTIS	A TIS instruction exists at *i* or after.
iMEM	*i* is the first MEM and not blocked
rMEM	*iMEM* exists at *i* or before
iNOP	*i* is the next op in the queue
rNOP	*iNOP* exists at *i* or before
iAIO	*i* is the first AIO and not blocked
rAIO	*iAIO* exists at *i* or before
iALU	*i* is allocated to the ALU pipe
rALU	*iALU* exists at *i* or before
VNI	*i* is valid (enabled) and not issued
ISSUED	*i* is issued on this clock

4.3 Control

The fundamental signals from each buffer stage are combined to form seven busses: *PCM(2:0)*, *iNOP(2:0)*, *iALU(2:0)*, *iAIO(2:0)*, *iMEM(2:0)*, *VNI(2:0)*, and *ISSUED(2:0)*. Bit 0 of each corresponds to the B0 buffer stage, bit 1 corresonds to the M0 buffer stange, and bit 2 corresponds to the M1 buffer stage. These signals are used throughout the dispatch as control signals.

The only system–level control interaction the dispatch has is with the memory pipe. The following signals are used for flow control:

Name	Description
NOFETCH	DSP tells MEM to not write the incremented PC: stall
PCC	MEM tells DSP that fetching can continue
CONT	MEM tells DSP that the branch was not taken

When there is no instruction allocated to a pipe, the *NOP* flag for that pipe is set to logic 1, disabling the don't–care instruction as it propagates down the pipe.

4.4 Logic Development

The logic used in the dispatch unit was firt prototyped in C++. Here is the code engine which demonstrates the logic.

buffer.h

```
#ifndef BUFFER_H
#define BUFFER_H

#include "types.h"
#include "bufnode.h"

#define M1 2
#define M0 1
#define B0 0

#define NUMSTAGE (M1+1)

class BUFFER
{
  public:
  BUFFER ();
  ~BUFFER ();

  void Set (int i, ITYPE i);

  void Show (void);

  int iSum (int n);
  int rSum (int n);

  bool iALU (int n);
  bool rALU (int n); /* -r does include the current node */

  bool iTIS (int n);
  bool fTIS (int n);

  bool iNOP (int n);
  bool rNOP (int n);

  bool iAIO (int n);
  bool rAIO (int n);

  bool iMEM (int n);
  bool rMEM (int n);
  BUFNODE **buf;

    private:
    bool rBLOCK (int n);
};
```

```
#endif
```

buffer.cc

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#include "buffer.h"

/* ALU is a subset of SIO */

void BUFFER::Set (int i, ITYPE p)
{
  *buf[i] = p;
}

void BUFFER::Show (void)
{
  int i;
  for (i = NUMSTAGE - 1; i >= 0; i--)
    {
      buf[i]->Show ();
      printf (" ");
    }
  printf ("\n");
}

BUFFER::BUFFER ()
{
  buf = new (BUFNODE *)[NUMSTAGE];
  buf[0] = new BUFNODE ("B0");
  buf[1] = new BUFNODE ("M0");
  buf[2] = new BUFNODE ("M1");

}

BUFFER::~BUFFER ()
{
  delete buf[2];
  delete buf[1];
  delete buf[0];
  delete[]buf;
}
```

```
int BUFFER::iSum (int n)
{
}

int BUFFER::rSum (int n)
{
}

bool BUFFER::rBLOCK (int n) /* blocked at or before n ? */
{
  if (n < 0)
    return 0;
  else if (buf[n]->Blocking () && buf[n]->Enabled ())
    return true;
  else
    return rBLOCK (n - 1);
}


bool BUFFER::iTIS (int n)
{
  return
    !rBLOCK (n - 1)
    &&
    buf[n]->Enabled ()
    &&
    (buf[n]->Type () == TIS)
    ;
}

bool BUFFER::fTIS (int n) /* is there a TIS op. at or beyond n ? */
{
  if (n >= NUMSTAGE)
    return false;
  else
    {
      if (iTIS (n))
return true; /* seek to first TIS op */
      else
return fTIS (n + 1);
    }

}

bool BUFFER::iNOP (int n)
{
```

```
  return
    buf[n]->Enabled ()
    &&
    (buf[n]->Type () == NOP)
    &&
    !rNOP (n - 1)
    &&
    !rAIO (n - 1)
    &&
    !rALU (n - 1)
    &&
    !rMEM (n - 1)
    ;
}

bool BUFFER::rNOP (int n)
{
  if (n < 0)
    return false;
  else
    {
      if (iNOP (n))
return 1; /* seek to first NOP */
      else
return rNOP (n - 1);
    }
}


bool BUFFER::iAIO (int n) /* is this the AIO op ? */
{
  return
    !rBLOCK (n - 1)
    &&
    buf[n]->Enabled ()
    &&
    (
      iNOP (n)
      ||
      (
(buf[n]->Type () == TIS) /* TIS ...  */
||
( /* or... */
  (buf[n]->Type () == ALU) /*  ALU */
  &&
  !fTIS (n) /* no forward enabled TIS ops */
```

```
)
      )
    )
    &&
    !rAIO (n - 1) /* and there are no previous AIO ops */
    ;
}

bool BUFFER::rAIO (int n) /* are there any AIO ops at or previous to n ? */
{
  if (n < 0)
    return false;
  else
    {
      if (iAIO (n))
return 1; /* seek to first AIO op */
      else
return rAIO (n - 1);
    }
}

bool BUFFER::iALU (int n) /* is this the primary ALU op ? */
{
  return
    !rBLOCK (n - 1)
    &&
    buf[n]->Enabled ()
    &&
    (
      iNOP (n)
      ||
      (
(buf[n]->Type () == ALU) /* this is type ALU */
&&
!iAIO (n) /* and this is not the AIO op */
      )
    )
    &&
    !rALU (n - 1) /* and ALU was not allocated before n */
    ;
}

bool BUFFER::rALU (int n) /* are there any ALU ops at or previous to n ? */
{
  if (n < 0)
    return false;
```

```
  else
    {
      if (iALU (n))
return 1; /* seek to first ALU op */
      else
return rALU (n - 1);
    }
}

bool BUFFER::iMEM (int n) /* is this the "primary" MEM op ? */
{
  return
    !rBLOCK (n - 1)
    &&
    buf[n]->Enabled ()
    &&
    (
      iNOP (n)
      ||
      (buf[n]->Type () == MEM) /* MEM operation */
    )
    &&
    !rMEM (n - 1) /* and there are no previous MEM ops */
    ;
}

bool BUFFER:: rMEM (int n) /* are there any MEM ops at or previous to n ? */
{
  if (n < 0)
    return false;
  else
    {
      if (iMEM (n))
return 1;
      else
return rMEM (n - 1);
    }
}
```

4.5 FPGA Utilization

Resource	Used	Total	Percent
Occupied CLBs	149	196	76%
Bonded I/O Pins	98	112	87%
F and G Function Generators	219	392	55%

H Function Generators	74	196	37%
CLB Flip Flops	58	392	14%
IOB Input Flip Flops	0	112	0%
IOB Output Flip Flops	0	112	0%
3-State Buffers	0	448	0%
3-State Half Longlines	0	56	0%
Edge Decode Inputs	0	168	0%
Edge Decode Half Longlines	0	16	0%

4.6 Timing Data

```
Limit  Actual  Points
(ns)   * (ns)  Missed   Specification
------  ------  -------  -------------
<auto>   227.0    0/58   DEFAULT_FROM_FFS_TO_FFS=FROM:ffs:TO:ffs
<auto>    40.8    0/101  DEFAULT_FROM_PADS_TO_FFS=FROM:pads:TO:ffs
<auto>   223.4    0/54   DEFAULT_FROM_FFS_TO_PADS=FROM:ffs:TO:pads
```

The dispatch has a very long FF to FF delay, and this is probably because of the long logic path which goes through the three buffer logic blocks. If there was more time, the logic could have been optimized.

Chapter 5

Master & Slave ALU Pipe Design

5.1 Introduction

This part of the report includes the basic features of arithmetic logic unit together with I/O (AIO in breif). The AIO is mainly responsible for doing simple calculations, reading values from register file, writing the result back to register file and also doing some character manipulations (includes reading and writing) through dumb terminal. This AIO expects inputs from dispatch, memory pipe, ALU pipe, register file, usart and also from switches and passes the outputs to register file, ALU, memory pipe and to dumb terminal.

5.2 Components

The AIO/ALU pipes contain: ipads, ibuf, bufgs, flip-flops, logic gates, multiplexers, decoders, 8-bit built-in adder, obuf, opad and the TIS.

5.3 Instructions Handled & Format

The following notation convention has been defined:

Notation	Meaning
D	destination register
S	source register
SB	set bit
TBR	one bit signal from TIS
RDA	one bit signal from TIS
DT	dumb terminal
IMM3	three bit immediate field
D[...]	some specified bits of the register file

Instruction	Dest	Source 1	Source 2
`add`	D	S	S
`sub`	D	S	S
`and`	D	S	S
`or`	D	S	S
`addc`	D	S	S + Carry
`not`	D		S
`slt`	SB	S	S
`seq`	SB	S	S
`iotr`	D	TBR	
`iora`	D	RDA	
`iot`	DT	S	

`ior`	D	DT	
`lru`	D[...]		IMM3
`sll`	D	S	IMM3
`srl`	D	S	IMM3
`sra`	D	S	IMM3

5.4 Inputs

Signal Name	Description	Size	Origin
WR1	write enable one	1	ALU pipe
WR2	write enable two	1	Memory pipe
Clock	clock	1	System
SET2ENS2M	set enable to master	1	ALU pipe
NOPALU	ALU inst. is NOP	1	Dispatch
CARRYS2M	carry bit to master	1	ALU pipe
NOPAIO	AIO inst. is NOP	1	Dispatch
SETS2M	set bit to master	1	ALU pipe
CYENS2M	carry enable bit to master	1	ALU pipe
INSTAIO	inst. to AIO	12	Dispatch
RR0	data from reg. file	8	Reg. file
RR1	data from reg. file	8	Reg. file
S0	TIS clock divider	1	Switch
S1	TIS clock divider	1	Switch
TXRDY	transmitter ready	1	USART
RXRDY	receiver ready	1	USART
WS1	write select 1	3	ALU pipe
WS2	write select 2	3	ALU pipe

5.5 Outputs

Signal Name	Description	Size	Destination
WR0	write enable	1	Reg. file
CARRYM2S	carry master to slave	1	ALU pipe
TRAP	instruction trap	1	Mem pipe
SET	set bit	1	Mem pipe
RS0	register read select	3	Reg. file
RS1	register read select	3	Reg. file
WS0	register write select	3	Reg. file
WS1	register write select	3	Reg. file
WD0	register write data	8	Reg. file

USART_WRB	¬ WR	1	USART
USART_SCK	slow clock	1	USART
USART_CDB	C/¬ D	1	USART
USART_RDB	¬ RD	1	USART
USART_RESET	reset	1	USART
D_USART_OUT	data	8	USART

5.6 Pipeline Design

The main schematic of the AIO pipe is broken into three different pipeline stages.

5.6.1 First Pipeline Stage

Input *INST(12), NOP*
Output *INST(12), NOP*

The first pipeline stage consists of 13 one-bit "fd" flip-flops. It contains 12 bit instructions and one bit nop. It is to be mentioned here that the flip-flops are postive edge triggered. After the instructions and the nop bits are latched through the flip-flop, it (the 12 bit instruction) is forked into two different 6 bit buses; the first one containing the lower six bits and the other one contaning the upper six bits. The least significant six bit bus is used for selecting two three bit source registers from the register file. The upper six bit bus is used for partial I/O decoding.

Partial Decoding

Input *INST(6), NOP, REGS*
READTIS_OUT, TISDATA_OUT, IOCS_OUT, DISABLE_OUT,Output

This partial IO decoding is responsible for checking the `iot` instruction. In order to check for the IOT instruction, the most significant three bits of the instructions need to true (in this case the signal is named as SEVEN). This SEVEN signal is then used to generate the *RDA_TBR_OUT* signal (true if any of the `iotr` or `iora` instructions is true) which will be used as an IO flag in the next cycles. Also the `ior` instruction is tested at this stage and in accordance with that *READTIS_OUT* signal is generated; when high it indicates that the AIO pipe is executing the `ior` instruction. Last of all, this block checks for the `iot` instruction. If `iot` returns true, a couple of things happen at the same time which are following:

- If true, it generates active low *R_WB_OUT* signal for one clock pulse and it gets sent to TIS.
- If true, it also generates active high *IOCS* signal for one clock cycle and it also gets sent to TIS. When the AIO pipe executes the `iot` instruction, it is transmitting some data to TIS. In order to write something to TIS two things have to happen at the same time: *IOCS* goes high for one clock pulse and also *RWB* low for one pulse.
- If true, it allows the data (in this case coming from the register file) to go to TIS through tri–state buffer provided that AIO pipe is not executing a NOP instruction. In this case the *NOP* bit is working as a selector for 2 to 1 MUX. The reason is that even though the AIO pipe is executing the *NOP* instruction, the instruction bits (which are garbage at this point) may contain an `iot` instruction.
- If true, it generates *DISABLE_OUT* along with *NOP* which will be passed to the next pipe meaning the AIO will not do anything for the next two clock cycles in the second and third pipeline stages.

Destination Register Selection

Input *INST(5), NOP*
Output *DestRegSel*

This block is responsible for selecting the destination register for any instruction. For `add`, `sub`, `and`, `or`, `addc` and `not` instructions the destination register is specified in bits 7, 8, and 9 of the instruction; for set instrutions there is not destination register; for `iotr, iora, ior, lru, sll, srl, sra` instructions the destination register is specified in bits 3, 4, and 5 of the instruction. So it can easily be seen that the destination register can be selected in two different ways: through bit seven, eight and nine and also through bits three, four and five and in order to do this, three 4 to 1 muxes were chosen.

- S0 is low when `add, sub, and, or, addc` and `not` instructions are executed
- S0 is high when `iotr, iora, ior` and `lru` instructions are executed
- S1 is high when `nop` or `iot` instruction is executed

Based on this selection inputs, the appropriate bits are selected for destination registers.

5.6.2 Second Pipeline Stage

Input *REGDATA0(8), REGDATA1(8), INST(8)*
RDSEL(8), NOP_IOT, RDA_TBR, TIS_READ
Output *same*

The inputs are latched to "fd" flip–flops on the rising edge of the next clock and the outputs are used for doing arithmetic operations.

Operation

Input *OPCODE(3), DATA0(8), DATA1(8), OPEXT(3)*
IMM(3), DISABLE, CarryC, SetWoTIS
Output *AluOut(8), NIS, RDA_S, TBR_S, RTYPE*

This block is solely responsible for doing the arithmetic operations. I will go over each instruction and explain the related hardware. Also, the following is the select logic which is used to select the appropriate input:

TBR_S goes high when `iotr` instruction is being executed and passed to the next stage. *RDA_S* goes high when `iora` instruction is being executed and passed to the next stage. *TRAP* goes high when ALU and AIO both are executing set instructions. *RTYPE1* goes high when `add, sub, and, or, not, addc` instructions are executed. *NIS* goes high if the disable bit is high from the last clock cycle or the AIO pipe is executing the `set` instruction. This gets passed to the next stage and nothing will happen at that stage. *RYTPE* goes high when *RTYPE* goes high or when any one of LRU, SRL, SLL, SRA signals goes high. *SET_E* goes high when either one of the set instructions (`seq` or `slt`) is true and the AIO is not executing the DISABLE instruction from the past clock cycle. Select goes high when the AIO executes sub or set instructions.

For `add, sub` and `addc` instructions, a built–in 8 bit adder is used which expects two eight–bit inputs, one carry bit. It generates one carry–out bit and an 8–bit sum. For an `add` instruction, the carry–in is set to zero, for `sub` it is one and for the `addc` instruction the previous is carry (`CY`) is used and this is implemented by using one 4 to 1 mux. For `add` and `addc` instructions, two eight bit data busses go directly to the input of the adder; however, for the `sub` instruction the second operand is inverted and then passed to the adder. The result is passed through one 8 by 8 to 1 mux.

For `and`, `or` and `not` instructions, the first operand (which is the eight bit data) is passed through AND, OR and INVERTERS and the output is passed to the 8 bit 8 to 1 MUX.

For the `lru` instruction, the first six bits of the first operand is latched with the least two significant bits of the three bit immediate field. For the shifter, the instructions are tested first whether it is a shift instruction or not. Based on this, the input data gets passed to the shiftr block along with the three bit opcode text and also with three bit immediate field. The shifter basically is responsible for the shift left and for shift arithemetic right. The shiftout data and the LRU_OUT data is then passed through two tri-state buffers where LRU_OUT, SHIFT flags (determined from the instruction) are used to select the right output and hence this output is passed to the 8 bit 8 to 1 mux.

The set instruction is different compared to other instructions; to implement the set instruction, it has to do subtraction and based on the output of the subtraction, the set bit is set or not set. For the `seq` instruction, the output of the adder is checked whether it is equal to zero or not. If true, it will allow the mux to select high value for the set flip–flop. This flip–flop is enabled by SET_E signal. For the `slt` instruction, the most significant bit of the adder output is checked. One sets the set flip–flop. When the AIO pipe is executing the set instruction it will generate the NIS signal for the next stage to indicate that the AIO pipe wont do anything for the next stage.

After `add, sub, and, or, addc, shift, lru, set` instructions are executed, the three bit opcode is used to select the correct input from the eight bit eight to one mux and the output of the mux is propagated for the next pulse as aluouput.

5.6.3 Third Pipeline Stage

Input	*RTYPE, RDA_TBR, READTIS, NIS, ALUIN(8), DESTREGS(3)*
Output	*ALUOUT(8), DISABLE, TIS_RCV, IOSET, DestReg(3), RegOp*

In this stage, the inputs are stored in "fd" flip-flop and they are latched in the next clock cycle.

Write Back

Input	*TIS_RCV, R_TYPE, IOSET, DISABLE, Clock, RDA* *TBR, RDA_S, TBR_S*
Output	*FinalData(8), WE, IOCS, RWB, IOR*

This stage is writes back the result to the register. Two kinds of data are manipulated here: one coming from the ALU operation and the other coming from the TIS. These two 8 bit data are passed through 4 to 1 muxes. The select inputs for the muxes are the following togetherwith their values:

S0 goes high when IOTR, IORA, DISABLE bit is high or when both TIS_RCV, which determines IOR instructions, and RTYPE signals are low. S1 goes high when TIS_RCV or DISABLE is high.

If `add, sub, and, or, not, lru, sll, sra, srl,` or `addc` instructions are executed, then the aludata is gets written to the destination register; if `iotr` or `iora` instructions are executed, then 0000 0001 gets written to the destination register, if the `ior`

instruction gets executed couple of things happen at the same time:

- Generates the active high *IOCS* flag for one clock pulse
- Generates the active high *RWB* signal for one clock pulse. (It is to be menioned here that if the AIO pipe is executing the `ior` instrucntion, that means it has to deal with the TIS and the TIS will be responsible for receiving the characters from the Dumb terminal and in order to do that both the *IOCS* and *RWB* signals will be high one clock pulse)
- If true, then it puts the *TISDATA* to the destination register.

5.6.4 TRAP Generator

The trap gets generated in three different ways and will be discussed here briefly:

Set Collision

When both the Master and Slave ALU are executing the `set` instruction, the *TRAP* signal goes and it gets passed on to the Memory Pipe which will stall the `PC`.

IO Collision

There is a possibily that the ALU will may execute the `iot` instrucion in the first clock cycle and the `ior` instrucntion in the third clock cycle. If that is the case, it can be seen that both these two instructions will try to overwrite the *R_WB* signal (*IOT* causes the *RWB* to go low and *IOR* causes the *RWB* to go high) and this will cause unexpected behavior. The *TRAP* signal activates when this occurs, to alert the programmer to the error.

5.6.5 Final Set

This is used to set the set bit; if any of the set bits is high (either from Master ALU or from Slave ALU), the *FINAL_SET* gets high.

5.6.6 Final Carry

To set the final carry for the ALU, we pass the carry bit through fdce flip-flop which is enabled by *ENABLE* signal. This enable signal gets set when the master ALU is executing `add`, `sub` or `addc` (beacause these three instructions will set the carry after the operation) and there is no NOP in the master ALU or when the slave ALU is executing the three instructions `add, sub, or addc`, and the slave is also not executing no `nop` instructions. This enable signal delayed by two clock cycles because the carry gets generated in second clock cycle of the instruction.

5.6.7 TIS_RESET

TIS gets reset three cycles after the system reset and it stays high for two clock pulses.

5.7 FPGA Utilization

5.7.1 Master ALU Pipe

Resource	Used	Total	Percent
Occupied CLBs	187	196	95%
Bonded I/O Pins	83	112	74%
F and G Function Generators	266	392	67%
H Function Generators	60	196	30%
CLB Flip Flops	132	392	33%
IOB Input Flip Flops	0	112	0%
IOB Output Flip Flops	0	112	0%
3-State Buffers	40	448	8%
3-State Half Longlines	32	56	57%
Edge Decode Inputs	0	168	0%
Edge Decode Half Longlines	0	32	0%
CLB Fast Carry Logic	6	196	3%

5.7.2 Slave ALU Pipe

ALU without TIS is pretty much the same thing as AIO pipe except it does not include any TIS or any corresponding TIS signal. Other than that, the arithemtic operation and the decoding unit is the same.

Resource	Used	Total	Percent
Occupied CLBs	118	196	60%
Bonded I/O Pins	54	112	48%
F and G Function Generators	135	392	34%
H Function Generators	26	196	13%
CLB Flip Flops	56	392	14%
IOB Input Flip Flops	0	112	0%
IOB Output Flip Flops	0	112	0%

3-State Buffers	24	448	5%
3-State Half Longlines	16	56	28%
Edge Decode Inputs	0	168	0%
Edge Decode Half Longlines	0	32	0%
CLB Fast Carry Logic	6	196	3%

5.8 Timing Data

5.8.1 Master ALU Pipe

```
Limit  Actual  Points
(ns)   * (ns)  Missed   Specification
------  ------  -------  -------------
<auto>   119.9    0/139  DEFAULT_FROM_FFS_TO_FFS=FROM:ffs:TO:ffs
<auto>    50.3    0/161  DEFAULT_FROM_PADS_TO_FFS=FROM:pads:TO:ffs
<auto>    79.1    0/34   DEFAULT_FROM_FFS_TO_PADS=FROM:ffs:TO:pads
```

5.8.2 Slave ALU Pipe

```
Limit  Actual  Points
(ns)   * (ns)  Missed   Specification
------  ------  -------  -------------
<auto>   118.4    0/28   DEFAULT_FROM_FFS_TO_FFS=FROM:ffs:TO:ffs
<auto>    62.6    0/94   DEFAULT_FROM_PADS_TO_FFS=FROM:pads:TO:ffs
<auto>    90.6    0/21   DEFAULT_FROM_FFS_TO_PADS=FROM:ffs:TO:pads
```

Chapter 6

Memory Pipe Design

The Memory Pipe is a three stage pipeline, and resides in FPGA 3. It handles all instructions that access memory or modify the `PC`. In order to access 8k of data memory, and 16K of Instruction Memory, 14 bit addresses are needed. Because $\mathcal{RRISC}$ has only eight bit data words, and 8 general purpose registers, rather than use a register pair for the stack pointer and traditional loads/stores, two special 14–bit registers are used, `SP`, the Stack Pointer, and `AR`, the Address Register. The Memory Pipe is fully responsible for the Program Counter.

6.1 Instructions

The memory pipe handles all instructions that access data memory, either of the two special registers, `AR` and `SP`, and jumps/branches. In addition, `lrl` (load register low) is handled by the memory pipe. While `lrl` is not consistent with the other instructions in the pipe, it allows the Dispatch Unit to determine the appropriate pipe for each instruction based on only the instruction's MSB.

6.1.1 General Register Instructions

Instruction	Descriptions
`lrl`	load the lower 6 bits of a general purpose register with an immediate value, zero the upper 2 bits

6.1.2 Memory Access Instructions

Instruction	Descriptions
`lw`	read memory word into general purpose register
`sw`	store general purpose register in memory

6.1.3 Stack Instructions

Instruction	Descriptions
`pushr`	push a general purpose register onto the stack
`pushh`	push the upper 6 bits of the `AR` onto the stack
`pushl`	push the lower 8 bits of the `AR` onto the stack
`popr`	pop the top of the stack into a general purpose register
`poph`	pop the top of the stack into the upper 6 bits of the `AR`
`popl`	pop the top of the stack into the lower 8 bits of the `AR`
`artsp`	initialize the `SP` with the value of the `AR`
`sptar`	move the `SP` to the `AR`

6.1.4 Address Register Instructions

Instruction	Descriptions
`laru`	loads the upper 6 bits of the `AR` with the contents of a general purpose register
`larl`	loads the lower 8 bits of the `AR` with the contents of a general purpose register

6.1.5 Jump/Branch Instructions

Instruction	Descriptions
`bs`	branch on `SET` == 1
`bns`	branch on `SET` == 0
`ret`	return to stored `PC`
`jn`	jump to within +/- 128 instructions
`jaln`	jump to within +/- 128 instructions, store the jump `PC` in `AR`
`jf`	jump anywhere in Instruction Memory
`jalf`	jump anywhere in Instruction Memory, store the jump `PC` in `AR`

6.2 Interface

The Memory Pipe uses 94 I/O Pins. Of these, 49 are inputs, 37 are ouptuts, and 8 are bidirectional.

6.2.1 Input signals to the MEMORY PIPE

System Signals

Signal Name	Description
CLK	system clock
TRAP	stalls the PC until system reset, used in case of multiple writes to same location

From DISPATCH UNIT

Signal Name	Description
INSTMEM(12)	an instruction issued from the Dispatch Unit stripped of its most significant bit
PCDIS(14)	the PC of the instruction being issued, used

	to compute jump addresses
NOPMEM	indicates that the current INSTMEM is not valid and should not be executed
NOF	indicates the instruction buffer cannot hold two more instructions, stalling the `PC`

From REGISTER FILE

Signal Name	Description
RR4(8)	8 bits of read data

From DATA MEMORY

Signal Name	Description
MDAT(8)	8 bits of read data

From the ALU PIPES

Signal Name	Description
SET	used for branch condition checks

6.2.2 Output Signals from the MEMORY PIPE

To DISPATCH UNIT

Signal Name	Description
PCC	indicates a `PC` modifying instruction (jump) has cleared the memory pipe, and instruction flow can continue

To REGISTER FILE

Signal Name	Description
RS4(3)	read register select
WS2(3)	write register select
WRE2	write register enable
WD2(8)	write register data

To DATA MEMORY

Signal Name	Description
DADDR(13)	data address selection
MDAT(8)	data to be written to memory
MWE	memory write enable

To INSTRUCTION MEMORY

Signal Name	Description
PC(14)	the program counter, with its upper 12 bits as an index to two instructions in memory

6.3 Implementation

The Memory Pipe is a three stage pipeline. Input Flip-Flops are used as the first stage pipeline registers to store the instruction to be executed and its accompanying state information (`PC`, `SET`, *NOP* and *NOF* flags). All pipeline and special purpose registers are rising edge triggered on the system clock. Full instruction decoding is performed by the Control Unit in the first stage. Memory and jump address calculations, and Register File reads are also completed in the first stage. Data Memory accesses occur in the second stage. The third stage is used for Register File writeback. In the Control Unit, most multiplexor controls are implemented using 16X1 ROMs, and write enable signals for registers and memory are generated by hardwired controls. Write enable signals are also conditioned with the *NOP* flag, meaning no register or memory writes occur on a *NOP* in the instruction stream. The following is a more specific description of what is happening in each stage for each instruction:

General Register Instruction

Instructions `lrl`

Stage 1	Instruction Decode Extend Immediate Field
Stage 2	
Stage 3	Write Destination Register

Memory Access Instructions

Instructions `lw`, `sw`

Stage 1	Instruction Decode
	Compute Memory Address
	Read General Register (`sw`)
Stage 2	Access Data Memory
Stage 3	Write Destination Register (`lw`)

Stack Instructions

Instructions `pushr`, `pushh`, `pushl`, `popr`, `poph`, `popl`, `artsp`, `sptar`

Stage 1	Instruction Decode
	Load `SP`, `AR` (`artsp`, `sptar`)
	Read General Register (`pushr`)
	Increment/Decrement `SP` (all but `artsp`, `sptar`)
Stage 2	Access Data Memory
	Write to `AR` (`poph`, `popl`)
Stage 3	Write to General Register (`popr`)

Address Register Instructions

Instructions `laru`, `larl`

Stage 1	Instruction Decode
	Read General Register
	Write to `AR`
Stage 2	
Stage 3	

Jump/Branch Instructions

Instructions `bs`, `bns`, `jn`, `jaln`, `jf`, `jalf`

Stage 1	Instruction Decode
	Compute Jump Address
	Check Branch Condition
	Write New `PC`
	Write New `AR` (`jaln`, `jalf`)
Stage 2	Raise *PCC* (`PC` is now valid)
Stage 3	

Three 14 bit adders are used in the design. The first two are general purpose adders, and compute the memory and jump addresses respectively. The third is a special 'increment to the next even number' adder. This adder is used to increment the PC. See Section 4 for more details.

6.4 Program Counter

The PC is a 14 bit register used as the read address into Instruction Memory. Because Instruction Memory is arranged in two parallel banks of 8K instructions, only the upper 13 bits of the PC are used to address memory. Memory is also arranged such that two reads are always performed concurrently (in the event that one of the instructions is invalid, the Dispatch Unit sets a flag). The least significant bit of the PC is used only on jumps/branches when the destination address is odd. The PC is incremented by two on every clock cycle when the *NOF* flag from the Dispatch Unit is low. When *NOF* is high, the PC is stalled for that clock cycle. A special situation arises when the PC is odd as the result of a jump/branch. In this case, the PC is incremented by one. The PC is changed in the following situations:

$\mathtt{PC} \leftarrow \mathtt{PC} + 2$	on every clock cycle unless *NOF* is high
$\mathtt{PC} \leftarrow \mathtt{PC} + 1$	if PC odd as a result of a jump/branch
$\mathtt{PC} \leftarrow \mathtt{AR}$	on `jf`, `jalf`, `ret`
$\mathtt{PC} \leftarrow \mathtt{PC} + 8$ bit offset	on `bs`, `bns`, `jn`, `jaln`

6.5 Address Register

The 14 bit Address Register, AR, has two primary purposes. First, it is used as the base address calculation for loads and stores. Both load and store have six bit immediate fields, which are added to the AR when accessing memory. Additionally, the AR is used in conjunction with the PC on jumps, serving as the jump address or storing the return address. On a `jf` or `ret`, the AR is loaded into the PC. In a `jaln` or `jalf`, the PC and AR are swapped, thus saving the return address into the AR. In the case of multiple jump and links, as in a function calling several other functions, the AR is stored onto the stack using `pushh`, `pushl`, with the AR being restored by `popl` and `poph` upon return from the function calls. All AR modifying instructions are completed in the first stage with the exception of `poph` and `popl`. This means there is no latency between modifying the AR and using its value. The AR is changed by the following instructions:

`laru`	$\mathtt{AR}_{13:8} \leftarrow \mathtt{R1}_{5:0}$
`larl`	$\mathtt{AR}_{7:0} \leftarrow \mathtt{R1}$
`poph`	$\mathtt{AR}_{13:8} \leftarrow \mathtt{M}[\mathtt{SP}]$
`popl`	$\mathtt{AR}_{7:0} \leftarrow \mathtt{M}[\mathtt{SP}]$

`sptar`	$\texttt{AR} \leftarrow \texttt{SP}$
`jaln`	$\texttt{AR} \leftarrow \texttt{PC}$
`jalf`	$\texttt{AR} \leftarrow \texttt{PC}$

6.6 Stack Operations

All stack operations access Data Memory through the Stack Pointer (`SP`). In addition, all stack accesses automatically increment/decrement the `SP`. The stack can be located anywhere in Data Memory by loading the `AR` with the desired stack location then executing the artsp instruction. The stack grows up in memory.

`pushr`	$\texttt{M}[\texttt{SP}++] \leftarrow \texttt{R1}$
`pushh`	$\texttt{M}[\texttt{SP}++] \leftarrow \texttt{AR}_{13:8}$
`pushl`	$\texttt{M}[\texttt{SP}++] \leftarrow \texttt{AR}_{7:0}$
`popr`	$\texttt{AR} \leftarrow \texttt{M}[\texttt{SP}--]$
`poph`	$\texttt{AR}_{13:8} \leftarrow \texttt{M}[--\texttt{SP}]$
`popl`	$\texttt{AR}_{7:0} \leftarrow \texttt{M}[--\texttt{SP}]$
`artsp`	$\texttt{SP} \leftarrow \texttt{AR}$

6.7 Jumps / Branches

Jump/branch addresses are computed relative to *PCDIS*, which is their location in Instruction Memory. This is done because the `PC` is not incremented every clock cycle and is nearly always even. System definition also disallows more than one jump or branch instruction residing in the Dispatch Unit at once. This is to prevent corruption of the *PCDIS* value. Thus there must be at least two instruction between consecutive jumps or branches. Jump nears and branches have an 8 bit immediate field, and can jump to within +/- 128 instructions, while jump fars can jump anywhere within Instruction Memory. However, Jump fars require some overhead, because the `AR` must first be loaded (usually from a pointer table in memory), then the jump executed. Jump and Links (`jalX`), store *PCDIS* in the `AR`, and are used in conjuction with the `ret` instruction.

`bs`	$\texttt{PC} \leftarrow \texttt{PC} + \overleftarrow{\texttt{s}}\ \mathrm{Imm}_8 \Leftrightarrow \mathrm{SET} == 1$
`bns`	$\texttt{PC} \leftarrow \texttt{PC} + \overleftarrow{\texttt{s}}\ \mathrm{Imm}_8 \Leftrightarrow \mathrm{SET} == 1$
`jn`	$\texttt{PC} \leftarrow \texttt{PC} + \overleftarrow{\texttt{s}}\ \mathrm{Imm}_8$
`jaln`	$\texttt{PC} \leftarrow \texttt{PC} + \overleftarrow{\texttt{s}}\ \mathrm{Imm}_8 \parallel \texttt{AR} \leftarrow \texttt{PC}$
`jf`	$\texttt{PC} \leftarrow \texttt{AR}$
`jalf`	$\texttt{PC} \leftrightarrow \texttt{AR}$

`ret`   `PC` ← `AR`

6.8 FPGA Utilization

The Memory Pipe uses a large amount of available CLBs because of the large number of multiplexors needed. The instructions handled by the memory pipe are very diverse in terms of the location and size of the instruction fields used as offset calculation. Additionally, the 14 bit multiplexors associated with the `PC`, `AR`, and `SP` consume a large number of CLBs.

Resource	Used	Total	Percent
Occupied CLBs	180	196	91%
Bonded I/O Pins	94	112	83%
F and G Function Generators	248	392	63%
H Function Generators	58	196	29%
CLB Flip Flops	119	392	30%
IOB Input Flip Flops	26	112	23%
IOB Output Flip Flops	0	112	0%
3-State Buffers	0	448	0%
3-State Half Longlines	0	56	0%
Edge Decode Inputs	0	168	0%
Edge Decode Half Longlines	0	16	0%
CLB Fast Carry Logic	16	196	8%

6.9 Timing Data

The timing delays in the Memory Pipe are very small in comparison to the other FPGA's. Because I/O Flip-Flops are used on all inputs, the maximum input-pad-to-flip-flop delays are equal to the delay of an IBUF. The flip- flop-to-output-d delay is also small because all output signal pass through at most one gate before going to an OBUF. The flip-flop-to-flip-flop delay is somewhat larger because of the number of logic levels some signals must propagate through. The critical path here is the computaion of the destination address on branches and jump nears. The address must propagate through a 14 bit adder, one 4 to 1 mux, and two 2 to 1 muxes.

```
Limit  Actual  Points
(ns)   * (ns)  Missed   Specification
------  ------  -------  -------------
<auto>    86.1    0/144  DEFAULT_FROM_FFS_TO_FFS=FROM:ffs:TO:ffs
<auto>    47.8    0/231  DEFAULT_FROM_PADS_TO_FFS=FROM:pads:TO:ffs
<auto>    48.5    0/53   DEFAULT_FROM_FFS_TO_PADS=FROM:ffs:TO:pads
```

Chapter 7

Register File Design

The register file contains eight eight–bit general registers numbered 0 through 7. Reads from Register 0 give 0x00, and writes to Register 0 have no effect.

Each register consists of eight one bit registers. A one bit register is made out of a 1–bit ‘fd’ flip-flop. Each register is triggered on the negative edge of the system clock. Also, each regitser expects eight bit input data, system clock and the *WE* (write enable) select signal. Based on the value of the write enable, it performs the read or write command to the register.

It is to be mentioned here that our register file can perform five multiple reads and three multple write in one clock cycle. In order to do this, we had to make the write selection decoding logic which would expect three different write enable signals from three different pipes together with three different destination registers. The write select unit passes three eight bit data and the right control singal to Data select unit. Since there are eight different registers, we need eight different data select units. Each data select unit basically selects which data will get written to the register specified in the control.

In order to perform the read operation from the register file, we had to make five different eight bit 8 to 1 MUXs. The MUX selection inputs come from the three bit register select signals provided by each pipe.

While designing the register file we encountered many problems. The first problem was to make the whole thing fit into one FPGA. The first time we ran the xmake lasted for one hour and a half with 14% routing. Then we had to change the design specially the Data select unit. Like mentioned before we have got eight different data selection unit. When we did design this first, we made these eight data select unit out of gates; then we used sort of trial and error method by using seven data selection unit with gates and only one with tri state buffer and this happen to solve our problem. It can be seen from the following table that register file was almost full with one FPGA.

7.1 FPGA Utilization

Resource	Used	Total	Percent
Occupied CLBs	196	196	100%
Bonded I/O Pins	93	112	83%
F and G Function Generators	380	392	96%
H Function Generators	128	196	65%
CLB Flip Flops	56	392	14%
IOB Input Flip Flops	0	112	0%
IOB Output Flip Flops	0	112	0%
3-State Buffers	24	448	5%
3-State Half Longlines	16	56	28%
Edge Decode Inputs	0	168	0%
Edge Decode Half Longlines	0	16	0%

7.2 Timing Data

```
(ns)    * (ns)  Missed   Specification
------  ------  -------  -------------
<auto>    19.7    0/56   DEFAULT_FROM_FFS_TO_FFS=FROM:ffs:TO:ffs
<auto>    63.0    0/112  DEFAULT_FROM_PADS_TO_FFS=FROM:pads:TO:ffs
<auto>    61.1    0/40   DEFAULT_FROM_FFS_TO_PADS=FROM:ffs:TO:pads
```

Chapter 8

Superscalar Performance

Program	$\mathcal{RRISC}$ Superscalar (inst/cycle)	$\mathcal{RRISC}$ non–Superscalar (inst/cycle)	% Improvement
Matrix Add/Sub	1.13	0.90	26%
Matrix Multiply	0.55	0.40	38%
Sort (no I/O)	0.75	0.65	15%
I/O	0.43	0.43	0%
Ideal Case	1.96	0.98	100%

Instructions per cycle were counted using a 16 bit counter triggering the logic analyzer every 64K clock cycles. A 16 bit adder with carry out was used to keep a running sum of the instructions issued by the Dispatch Unit. NOPs are not counted as instructions.

The design for counting the number of instructions per cycle was placed in the last FPGA (#5). The schematic and traces of the design follow this section.

Pure Superscalar Code is a sequence of test instructions that has no known application other than to demonstrate the potentially high throughput achievable by $\mathcal{RRISC}$.

Chapter 9

Macro Assembler

9.1 Introduction

A macro–assembler, MAD[1], was developed for the $\mathcal{RRISC}$ project. The purpose of the macro–assembler was to provide an easy way for the programmer to do commonly needed tasks without having to re–implement them each time in the DELA code. MAD was written in PERL. The Life and Menu programs were written using MAD.

9.2 MAD Flow

MAD goes through several stages before emitting DELA code:

1. Open input file, start reading input
2. Check code format, on error exit
3. Break high–level commands into low–level MAD commands
4. Verify individual commands and parameter ranges
5. Expand low–level MAD commands to DELA code

9.3 Language Definition

Notational Conventions

In the following R[1-3] is one of [a-gz]. X is a value in hexadecimal (0xff) or base-10 (54). X can also be a single character in single quotes ('z') which evaluates to its ASCII value. And all other strings are literal (part of the language). L is an assembler label. Each command is seperated by a semicolon ";".

Definition

Command	Meaning
`at X:`	Put the next instruction at location X in memory
`L:`	Assembler Label
`R = R + R`	Add two registers
`R = R - R`	Subtract two registers
`R = R or R`	OR two registers
`R = R and R`	AND two registers
`R = R +c R`	Add with carry
`R = not R`	Invert R
`set R1 < R2`	Set less than
`set R1 > R2`	Set less than
`set R1 == R2`	Set equal
`R = TBR`	Put TBR flag into LSB of R

[1] MAD is an acronym for "Macro Assembler for DELA"

<table>
<tr><td><code>R = RDA</code></td><td>Put RDA flag into LSB of R</td></tr>
<tr><td><code>send R</code></td><td>Send register R to the TIS</td></tr>
<tr><td><code>R = recv</code></td><td>Receive from TIS into R</td></tr>
<tr><td><code>R = X</code></td><td>Loads value X into R, possibly multiple instructions</td></tr>
<tr><td><code>R <<= X</code></td><td>Shift Left R by X</td></tr>
<tr><td><code>R >>a= X</code></td><td>Shift Right Arithmetic R by X</td></tr>
<tr><td><code>R >>= X</code></td><td>Shift Right Logical R by X</td></tr>
<tr><td><code>R = M[sX]</code></td><td>Load memory location AR + X into register R</td></tr>
<tr><td><code>M[sX] = R</code></td><td>Store R into memory location AR + X</td></tr>
<tr><td><code>AR = R1 || R2</code></td><td>Load AR: R1 to upper byte, and R2 to lower byte</td></tr>
<tr><td><code>AR = X(R1,R2)</code></td><td>Load X into AR, using R1 and R2 for temporary registers</td></tr>
<tr><td><code>push ar</code></td><td>Push AR onto the stack</td></tr>
<tr><td><code>pop ar</code></td><td>Pop AR off of the stack</td></tr>
<tr><td><code>push R</code></td><td>Push register R onto the stack</td></tr>
<tr><td><code>pop R</code></td><td>Pop register R off of the stack</td></tr>
<tr><td><code>AR = SP</code></td><td>Copy SP into AR</td></tr>
<tr><td><code>SP = AR</code></td><td>Sopy AR into SP</td></tr>
<tr><td><code>ret</code></td><td>Return from subroutine</td></tr>
<tr><td><code>nop</code></td><td>Insert nop into instruction stream</td></tr>
<tr><td><code>bs L</code></td><td>Branch Set to label L</td></tr>
<tr><td><code>bns L</code></td><td>Branch Not Set to label L</td></tr>
<tr><td><code>jmp L</code></td><td>Jump Near to label L</td></tr>
<tr><td><code>jal L</code></td><td>Jump–and–Link Near to label L</td></tr>
<tr><td><code>jmpfar</code></td><td>Jump Far to label L</td></tr>
<tr><td><code>jalfar</code></td><td>Jump–and–Link Far to label L</td></tr>
<tr><td><code>R = 0</code></td><td>Load <code>0x00</code> into register R</td></tr>
<tr><td><code>done</code></td><td>Infinite loop — halt</td></tr>
<tr><td><code>input R</code></td><td>Wait for TIS, then put character into R</td></tr>
<tr><td><code>output R</code></td><td>Wait for TIS, then output character from R</td></tr>
<tr><td><code>R++</code></td><td>Increment R</td></tr>
<tr><td><code>R--</code></td><td>Decrement R</td></tr>
<tr><td><code>R+=X</code></td><td>Increment R by X</td></tr>
<tr><td><code>R-=X</code></td><td>Decrement R by X</td></tr>
<tr><td><code>R = R</code></td><td>Copy register value</td></tr>
<tr><td><code>AR+=X</code></td><td>Imcrement AR by X</td></tr>
<tr><td><code>AR-=X</code></td><td>Decrement AR by X</td></tr>
</table>

9.4 Example

```
a = 1;
b = 'B';
g = b - a;
output g; ! output "A"
g = b + z;
```

```
output g; ! output "B"
g = b + a;
output g; ! output "C"

ar = 50(e, f);  ! initialize AR with decimal value 50
A = 'Y';
m[0] = a;       ! write $1 to memory location 50
a = 55;
m[1] = a;       ! write $1 to memory location 51
input g;
m[2] = g;       ! write key board ASCII value to memory location 50 + 2
!
! now print
!
g = m[0];
output g; ! output a "Y"
g = m[1];
output g; ! output a LUCKY "7"
g = m[2];
output g; ! output the last key user pressed
done;
```

Chapter 10

Individual Contributions

10.1 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.

10.2 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.

10.3 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.

10.4 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.

10.5 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.

10.6 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%

Chapter 11

Limits and Epilogue

11.1 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.

11.2 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 and hopes to pursue a Ph.D. in chicken farming. Asked about his future plans, Jeremy responded "I'm kind of tired, I think I might go to bed."

Chapter 12

DELA Definition

This is the DELA instruction format specification.

```
14 16     ! 16K X 16 bits memory


nothing      15, 13, 0;     ! bit 13, 14 & 15 are not used


opcode_4bit     12, 9, 0     ! Default opcode is 0

     add     0H
     sub     1H
     and     2H
     or      3H
     addc    4H
     not     5H
     lrl     8H
     lw      9H
     sw      0AH;


opcode_5bit     12, 8, 0     ! Default opcode is 0

     bs     1AH
     bns    1BH
     jn     1CH
     jf     1DH
     jaln   1EH
     jalf   1FH;


opcode_7bit     12, 6, 0     ! Default opcode is 0

     slt     30H
     seq     31H
     iotr    38H
     iora    39H
     iot     3AH
     ior     3BH
     lru     3CH
     sll     3DH
     srl     3EH
     sra     3FH
     pushr   58H
     larl    5AH
     popr    5CH
```

```
     laru    5EH
     popl    60H
     poph    61H
     ret     62H
     artsp   63H
     pushh   64H
     pushl   65H
     sptar   66H;

opcode_13bit     12, 0, 0     ! Default is 0
     nop0     19C0H
     nop1     19C1H
     nop2     19C2H
     nop3     19C3H
     nop4     19C4H
     nop5     19C5H
     nop6     19C6H
     nop7     19C7H
     nop8     19C8H
     nop9     19C9H
     nopa     19CAH
     nopb     19CBH
     nopc     19CCH
     nopd     19CDH
     nope     19CEH
     nopf     19CFH;


rd     8, 6, 0               ! Default Rd is $0

     d0     0H
     d1     1H
     d2     2H
     d3     3H
     d4     4H
     d5     5H
     d6     6H
     d7     7H;


rs     5, 3, 0               ! Default Rs is $0

     s0     0H
     s1     1H
     s2     2H
     s3     3H
```

```
     s4     4H
     s5     5H
     s6     6H
     s7     7H;

rt     2, 0, 0           ! Default Rt is $0

     t0     0H
     t1     1H
     t2     2H
     t3     3H
     t4     4H
     t5     5H
     t6     6H
     t7     7H;

imm6     5, 0, 0;          ! Default 6 bit immediate is 00H

imm3     2, 0, 0;           ! Default 3 bit immediate is 0H

jaddr     7, 0, 0;             ! Default jump address is 00H
```

Chapter 13

Software

(not included due to space considerations)

Chapter 14

Annotated Quicksim Traces

(not included due to space considerations)

Chapter 15

Full Schematics

(not included due to space considerations)