ELEC6027 - VLSI Design Project : Programmers Guide

Team R4

28$^{\text{th}}$ April, 2014

Todo list

Contents

1 Introduction

Lorem Ipsum...

2 Architecture

Lorem Ipsum...

3 Register Description

The processor consists of 16bit program counter and 8 general purpose registers of 16 bit each in the register file, 16 bit instruction register along with the Link register and AluOut.

Register File The 8x16 register file has two read ports and one write ports. The read port takes 3-bit address inputs, RS1 and RS2, each specifying one of the 8 GPRs as source operands. They read the 16 bit register values onto read data outputs RD1 and RD2. The write port takes a 3 bit address input Rw, a 16 bit input, WD. The output ports of the register file serve as inputs A and B of the ALU if selected by the two multiplexers. By convention, register R7 is used to store the stack pointer.

Link Register The Link Register is used to store the return address of the caller function. However, it is not a part of the General Purpose register file. For example, in this processor the RET instruction performs unconditional branch to the address which is stored in the link register (LR).

Program Counter The Program Counter (PC) is a 16bit register wherein the output of the PC points to the current instruction and the input of the PC shows the address of the next instruction. The branch instructions make use of the PC.

Instruction Register The instruction register of 16 bit is also a storage element that has a single read port. It takes a 16-bit instruction address input and reads the 16-bit data or instruction from that address onto the read data output (Rd).

AluOut It is a 16 bit register, AluOut stores the output of the operations performed in the ALU.

4 Instruction Set

The complete instruction set architecture includes a number of instructions for performing calculations on data, memory access, transfer of control within a program and interrupt handling. All instructions implemented by this architecture fall into one of 6 groups, categorized as follows:

- Data Manipulation - Arithmetic, Logical, Shifting
- Byte Immediate - Arithmetic, Byte Load
- Data Transfer - Memory Access
- Control Transfer - (Un)conditional Branching
- Stack Operations - Push, Pop
- Interrupts - Enabling, Status Storage, Returning

There is only one addressing mode associated with each instruction, generally following these groupings:

- Data Manipulation - Register-Register, Register-Immediate
- Byte Immediate - Register-Immediate
- Data Transfer - Base Plus Offset
- Control Transfer - PC Relative, Register-Indirect, Base Plus Offset
- Stack Operations - Register-Indirect Preincrement/Postdecrement
- Interrupts - Register-Indirect Preincrement/Postdecrement

4.1 General Instruction Formatting

<table>
<tr><th colspan="2">Instruction Type</th><th>Sub-Type</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>A1</td><td rowspan="2">Data Manipulation</td><td>Register</td><td colspan="5" rowspan="2">Opcode</td><td colspan="3">Rd</td><td colspan="3">Ra</td><td colspan="3">Rb</td><td colspan="2">X X</td></tr>
<tr><td>A2</td><td>Immediate</td><td colspan="3">Rd</td><td colspan="3">Ra</td><td colspan="5">imm4/5</td></tr>
<tr><td>B</td><td colspan="2">Byte Immediate</td><td colspan="5">Opcode</td><td colspan="3">Rd</td><td colspan="8">imm8</td></tr>
<tr><td>C</td><td colspan="2">Data Transfer</td><td>0</td><td>LS</td><td>0</td><td>0</td><td>0</td><td colspan="3">Rd</td><td colspan="3">Ra</td><td colspan="5">imm5</td></tr>
<tr><td>D1</td><td rowspan="2">Control Transfer</td><td>Others</td><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">0</td><td colspan="3" rowspan="2">Cond.</td><td colspan="8">imm8</td></tr>
<tr><td>D2</td><td>Jump</td><td colspan="3">Ra</td><td colspan="5">imm5</td></tr>
<tr><td>E</td><td colspan="2">Stack Operations</td><td>0</td><td>U</td><td>0</td><td>0</td><td>1</td><td>L</td><td colspan="2">X X</td><td colspan="3">Ra</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>F</td><td colspan="2">Interrupts</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td colspan="3">ICond.</td><td>1</td><td>1</td><td>1</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr>
</table>

Instruction Field Definitions

Opcode: Operation code as defined for each instruction
Rd: Destination Register
Ra: Source register 1
Rb: Source register 2
imm$N$: Immediate value of length N
Cond.: Branching condition code as defined for branch instructions
ICond.: Interrupt instruction code as defined for interrupt instructions
LS: 0=Load Data, 1=Store Data
U: 1=PUSH, 0=POP
L: 1=Use Link Register, 0=Use GPR

Pseudocode Notation

Symbol	Meaning
$\leftarrow$	Assignment
Result[x]	Bit x of result
Ra[x : y]	Bit range from x to y of register Ra
$<$	Numerically less than
$>$	Numerically greater than
$<<$	Logical shift left
$>>$	Logical shift right
$>>>$	Arithmetic shift right
Mem[*val*]	Data at memory location with address *val*
$\{x, y\}$	Contatenation of x and y to form a 16-bit value
!	Bitwise Negation

Use of the word UNPREDICTABLE indicates that the resultant flag value after operation execution will not be indicative of the ALU result. Instead its value will correspond to the result of an undefined arithmetic operation and as such should not be used.

4.2 ADD — Add Word

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	Rd			Ra			Rb			X	X

Syntax

ADD Rd, Ra, Rb   eg. ADD R5, R3, R2

Operation

$$\begin{aligned}
\mathrm{Rd} &\leftarrow \mathrm{Ra} + \mathrm{Rb} \\
\mathrm{N} &\leftarrow \text{if (Result} < 0\text{) then 1, else 0} \\
\mathrm{Z} &\leftarrow \text{if (Result} = 0\text{) then 1, else 0} \\
\mathrm{V} &\leftarrow \text{if (Ra>0 and Rb>0 and Result<0) or} \\
&\qquad\quad \text{(Ra<0 and Rb<0 and Result>0) then 1, else 0} \\
\mathrm{C} &\leftarrow \text{if (Result} > 2^{16} - 1\text{) or} \\
&\qquad\quad \text{(Result} < -2^{16}\text{) then 1, else 0}
\end{aligned}$$

Description

The 16-bit word in GPR[Ra] is added to the 16-bit word in GPR[Rb] and the result is placed into GPR[Rd].

Addressing Mode: Register-Register.

4.3 ADDI — Add Immediate

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	Rd			Ra			imm5				

Syntax

ADDI Rd, Ra, #imm5 eg. ADDI R5, R3, #7

Operation

$$\text{Rd} \leftarrow \text{Ra} + \#\text{imm5}$$

$$\text{N} \leftarrow \text{if (Result} < 0\text{) then 1, else 0}$$

$$\text{Z} \leftarrow \text{if (Result} = 0\text{) then 1, else 0}$$

$$\text{V} \leftarrow \text{if (Ra>0 and \#imm5>0 and Result<0) or}$$

$$\text{(Ra<0 and \#imm5<0 and Result>0) then 1, else 0}$$

$$\text{C} \leftarrow \text{if (Result} > 2^{16} - 1\text{) or}$$

$$\text{(Result} < -2^{16}\text{) then 1, else 0}$$

Description

The 16-bit word in GPR[Ra] is added to the sign-extended 5-bit value given in the instruction and the result is placed into GPR[Rd].

Addressing Mode: Register-Immediate.

4.4 ADDIB — Add Immediate Byte

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	Rd			imm8							

Syntax

ADDIB Rd, #imm8 — eg. ADDIB R5, #93

Operation

$$\mathrm{Rd} \leftarrow \mathrm{Rd} + \#\mathrm{imm8}$$

$$\mathrm{N} \leftarrow \text{if (Result} < 0\text{) then 1, else 0}$$

$$\mathrm{Z} \leftarrow \text{if (Result} = 0\text{) then 1, else 0}$$

$$\mathrm{V} \leftarrow \text{if (Rd>0 and \#imm8>0 and Result<0) or}$$

$$\text{(Rd<0 and \#imm8<0 and Result>0) then 1, else 0}$$

$$\mathrm{C} \leftarrow \text{if (Result} > 2^{16} - 1\text{) or}$$

$$\text{(Result} < -2^{16}\text{) then 1, else 0}$$

Description

The 16-bit word in GPR[Rd] is added to the sign-extended 8-bit value given in the instruction and the result is placed into GPR[Rd].

Addressing Mode: Register-Immediate.

4.5 ADC — Add Word With Carry

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Rd			Ra			Rb			X	X

Syntax

ADC Rd, Ra, Rb — eg. ADC R5, R3, R2

Operation

$Rd \leftarrow Ra + Rb + C$

$N \leftarrow$ if (Result < 0) then 1, else 0

$Z \leftarrow$ if (Result $= 0$) then 1, else 0

$V \leftarrow$ if (Ra>0 and (Rb+CFlag)>0 and Result<0) or

(Ra<0 and (Rb+CFlag)<0 and Result>0) then 1, else 0

$C \leftarrow$ if (Result $> 2^{16} - 1$) or

(Result $< -2^{16}$) then 1, else 0

Description

The 16-bit word in GPR[Ra] is added to the 16-bit word in GPR[Rb] with the added carry in set according to the Carry flag from previous operation, and the result is placed into GPR[Rd].

Addressing Mode: Register-Register.

4.6 ADCI Add Immediate With Carry

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	Rd			Ra			imm5				

Syntax

ADCI Rd, Ra, #imm5 eg. ADCI R5, R4, #7

Operation

$$\begin{aligned}
Rd &\leftarrow Ra + \#imm5 + C \\
N &\leftarrow \text{if (Result} < 0\text{) then 1, else 0} \\
Z &\leftarrow \text{if (Result} = 0\text{) then 1, else 0} \\
V &\leftarrow \text{if (Ra>0 and (\#imm5+CFlag)>0 and Result<0) or} \\
&\quad\quad \text{(Ra<0 and (\#imm5+CFlag)<0 and Result>0) then 1, else 0} \\
C &\leftarrow \text{if (Result} > 2^{16} - 1\text{) or} \\
&\quad\quad \text{(Result} < -2^{16}\text{) then 1, else 0}
\end{aligned}$$

Description

The 16-bit word in GPR[Ra] is added to the sign-extended 5-bit value given in the instruction with carry in set according to the Carry flag from previous operation, and the result is placed into GPR[Rd].

Addressing Mode: Register-Immediate.

4.7 NEG — Negate Word

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	Rd			Ra			X	X	X	X	X

Syntax

NEG Rd, Ra

eg. NEG R5, R3

Operation

Rd $\leftarrow$ 0 - Ra

N $\leftarrow$ if (Result $<$ 0) then 1, else 0

Z $\leftarrow$ if (Result $=$ 0) then 1, else 0

V $\leftarrow$ 0

C $\leftarrow$ 0

Description

The 16-bit word in GPR[Ra] is added to the 16-bit word in GPR[Rb] and the result is placed into GPR[Rd].

Addressing Mode: Register-Register.

4.8 SUB — Subtract Word

Format

15	14	13	12	11	10 9 8	7 6 5	4 3 2	1	0
0	1	0	1	0	Rd	Ra	Rb	X	X

Syntax

SUB Rd, Ra, Rb — eg. SUB R5, R3, R2

Operation

Rd $\leftarrow$ Ra - Rb

N $\leftarrow$ if (Result $<$ 0) then 1, else 0

Z $\leftarrow$ if (Result $=$ 0) then 1, else 0

V $\leftarrow$ if (Ra>0 and Rb>0 and Result<0) or

(Ra<0 and Rb<0 and Result>0) then 1, else 0

C $\leftarrow$ if (Result $> 2^{16} - 1$) or

(Result $< -2^{16}$) then 1, else 0

Description

The 16-bit word in GPR[Rb] is subtracted from the 16-bit word in GPR[Ra] and the result is placed into GPR[Rd].

Addressing Mode: Register-Register.

4.9 SUBI — Subtract Immediate

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	Rd			Ra			imm5				

Syntax

SUBI Rd, Ra, #imm5 — eg. SUBI R5, R3, #7

Operation

Rd $\leftarrow$ Ra - #imm5

N $\leftarrow$ if (Result $<$ 0) then 1, else 0

Z $\leftarrow$ if (Result $=$ 0) then 1, else 0

V $\leftarrow$ if (Ra>0 and #imm5>0 and Result<0) or

(Ra<0 and #imm5<0 and Result>0) then 1, else 0

C $\leftarrow$ if (Result $> 2^{16} - 1$) or

(Result $< -2^{16}$) then 1, else 0

Description

The sign extended 5-bit value given in the instruction is subtracted from the 16-bit word in GPR[Ra] and the result is placed into GPR[Rd].

Addressing Mode: Register-Immediate.

4.10 SUBIB — Subtract Immediate Byte

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	Rd			imm8							

Syntax

SUBIB Rd, #imm8 — eg. SUBIB R5, #93

Operation

$\text{Rd} \leftarrow \text{Rd - \#imm8}$

$\text{N} \leftarrow \text{if (Result} < 0\text{) then 1, else 0}$

$\text{Z} \leftarrow \text{if (Result} = 0\text{) then 1, else 0}$

$\text{V} \leftarrow \text{if (Rd>0 and \#imm8>0 and Result<0) or}$

$\quad\quad \text{(Rd<0 and \#imm8<0 and Result>0) then 1, else 0}$

$\text{C} \leftarrow \text{if (Result} > 2^{16} - 1\text{) or}$

$\quad\quad \text{(Result} < -2^{16}\text{) then 1, else 0}$

Description

The 8-bit immediate value given in the instruction is subtracted from the 16-bit word in GPR[Rd] and the result is placed into GPR[Rd].

Addressing Mode: Register-Immediate.

4.11 SUC Subtract Word With Carry

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	Rd			Ra			Rb			X	X

Syntax

SUC Rd, Ra, Rb eg. SUC R5, R3, R2

Operation

Rd $\leftarrow$ Ra - Rb - C

N $\leftarrow$ if (Result $<$ 0) then 1, else 0

Z $\leftarrow$ if (Result = 0) then 1, else 0

V $\leftarrow$ if (Ra>0 and (Rb-CFlag)>0 and Result<0) or

(Ra<0 and (Rb-CFlag)<0 and Result>0) then 1, else 0

C $\leftarrow$ if (Result $> 2^{16} - 1$) or

(Result $< -2^{16}$) then 1, else 0

Description

The 16-bit word in GPR[Rb] is subtracted from the 16-bit word in GPR[Rb] with the subtracted carry in set according to the Carry flag from previous operation, and the result is placed into GPR[Rd].

Addressing Mode: Register-Register.

4.12 SUCI Subtract Immediate With Carry

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	Rd			Ra			imm5				

Syntax

SUCI Rd, Ra, #imm5 eg. SUCI R5, R4, #7

Operation

Rd $\leftarrow$ Ra - #imm5 - C

N $\leftarrow$ if (Result $<$ 0) then 1, else 0

Z $\leftarrow$ if (Result $=$ 0) then 1, else 0

V $\leftarrow$ if (Ra>0 and (#imm5-CFlag)>0 and Result<0) or

(Ra<0 and (#imm5-CFlag)<0 and Result>0) then 1, else 0

C $\leftarrow$ if (Result $> 2^{16} - 1$) or

(Result $< -2^{16}$) then 1, else 0

Description

The 5-bit immediate value in instruction is subtracted from the 16-bit word in GPR[Ra] with the subtracted carry in set according to the Carry flag from previous operation, and the result is placed into GPR[Rd].

Addressing Mode: Register-Immediate.

4.13 CMP — Compare Word

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	X	X	X	Ra			Rb			X	X

Syntax

CMP Ra, Rb — eg. CMP R3, R2

Operation

$$\text{Ra - Rb}$$

$$N \leftarrow \text{if (Result} < 0) \text{ then 1, else 0}$$

$$Z \leftarrow \text{if (Result} = 0) \text{ then 1, else 0}$$

$$V \leftarrow \text{if (Ra>0 and Rb>0 and Result<0) or}$$
$$\text{(Ra<0 and Rb<0 and Result>0) then 1, else 0}$$

$$C \leftarrow \text{if (Result} > 2^{16} - 1) \text{ or}$$
$$\text{(Result} < -2^{16}) \text{ then 1, else 0}$$

Description

The 16-bit word in GPR[Rb] is subtracted from the 16-bit word in GPR[Ra] and the status flags are updated without saving the result.

Addressing Mode: Register-Register.

4.14 CMPI — Compare Immediate

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	X	X	X	Ra			imm5				

Syntax

CMPI Ra, #imm5 — eg. CMPI R3, #7

Operation

Ra - #imm5

$N \leftarrow$ if (Result $<$ 0) then 1, else 0

$Z \leftarrow$ if (Result $=$ 0) then 1, else 0

$V \leftarrow$ if (Ra>0 and #imm5>0 and Result<0) or

(Ra<0 and #imm5<0 and Result>0) then 1, else 0

$C \leftarrow$ if (Result $> 2^{16} - 1$) or

(Result $< -2^{16}$) then 1, else 0

Description

The sign extended 5-bit value given in the instruction is subtracted from the 16-bit word in GPR[Ra] and the status flags are updated without saving the result.

Addressing Mode: Register-Immediate.

4.15 AND Logical AND

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	Rd			Ra			Rb			X	X

Syntax

AND Rd, Ra, Rb　　　　eg. AND R5, R3, R2

Operation

$$\begin{aligned} Rd &\leftarrow Ra \texttt{ AND } Rb \\ N &\leftarrow \text{if (Result} < 0\text{) then 1, else 0} \\ Z &\leftarrow \text{if (Result} = 0\text{) then 1, else 0} \\ V &\leftarrow \text{UNPREDICTABLE} \\ C &\leftarrow \text{UNPREDICTABLE} \end{aligned}$$

Description

The logical `AND` of the 16-bit words in GPR[Ra] and GPR[Rb] is performed and the result is placed into GPR[Rd].

Addressing Mode: Register-Register.

4.16 OR — Logical OR

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	Rd			Ra			Rb			X	X

Syntax

OR Rd, Ra, Rb — eg. OR R5, R3, R2

Operation

$$\begin{aligned} Rd &\leftarrow Ra\ \texttt{OR}\ Rb \\ N &\leftarrow \text{if (Result} < 0\text{) then 1, else 0} \\ Z &\leftarrow \text{if (Result} = 0\text{) then 1, else 0} \\ V &\leftarrow \text{UNPREDICTABLE} \\ C &\leftarrow \text{UNPREDICTABLE} \end{aligned}$$

Description

The logical `OR` of the 16-bit words in GPR[Ra] and GPR[Rb] is performed and the result is placed into GPR[Rd].

Addressing Mode: Register-Register.

4.17 XOR — Logical XOR

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	Rd			Ra			Rb			X	X

Syntax

XOR Rd, Ra, Rb   eg. XOR R5, R3, R2

Operation

$$\begin{aligned} Rd &\leftarrow Ra \ \texttt{XOR}\ Rb \\ N &\leftarrow \text{if (Result} < 0\text{) then 1, else 0} \\ Z &\leftarrow \text{if (Result} = 0\text{) then 1, else 0} \\ V &\leftarrow \text{UNPREDICTABLE} \\ C &\leftarrow \text{UNPREDICTABLE} \end{aligned}$$

Description

The logical `XOR` of the 16-bit words in GPR[Ra] and GPR[Rb] is performed and the result is placed into GPR[Rd].

Addressing Mode: Register-Register.

4.18 NOT — Logical NOT

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	Rd			Ra			X	X	X	X	X

Syntax

NOT Rd, Ra — eg. NOT R5, R3

Operation

Rd ← `NOT` Ra

N ← if (Result $<$ 0) then 1, else 0

Z ← if (Result $=$ 0) then 1, else 0

V ← UNPREDICTABLE

C ← UNPREDICTABLE

Description

The logical `NOT` of the 16-bit word in GPR[Ra] is performed and the result is placed into GPR[Rd].

Addressing Mode: Register-Register.

4.19 NAND — Logical NAND

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	Rd			Ra			Rb			X	X

Syntax

NAND Rd, Ra, Rb — eg. NAND R5, R3, R2

Operation

$$\begin{aligned} Rd &\leftarrow Ra\ \texttt{NAND}\ Rb \\ N &\leftarrow \text{if (Result} < 0\text{) then 1, else 0} \\ Z &\leftarrow \text{if (Result} = 0\text{) then 1, else 0} \\ V &\leftarrow \text{UNPREDICTABLE} \\ C &\leftarrow \text{UNPREDICTABLE} \end{aligned}$$

Description

The logical `NAND` of the 16-bit words in GPR[Ra] and GPR[Rb] is performed and the result is placed into GPR[Rd].

Addressing Mode: Register-Register.

4.20 NOR — Logical NOR

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	Rd			Ra			Rb			X	X

Syntax

NOR Rd, Ra, Rb — eg. NOR R5, R3, R2

Operation

$$\begin{aligned}
\mathrm{Rd} &\leftarrow \mathrm{Ra}\ \texttt{NOR}\ \mathrm{Rb} \\
\mathrm{N} &\leftarrow \text{if (Result} < 0\text{) then 1, else 0} \\
\mathrm{Z} &\leftarrow \text{if (Result} = 0\text{) then 1, else 0} \\
\mathrm{V} &\leftarrow \text{UNPREDICTABLE} \\
\mathrm{C} &\leftarrow \text{UNPREDICTABLE}
\end{aligned}$$

Description

The logical `NOR` of the 16-bit words in GPR[Ra] and GPR[Rb] is performed and the result is placed into GPR[Rd].

Addressing Mode: Register-Register.

4.21 LSL — Logical Shift Left

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	Rd			Ra			0	imm4			

Syntax

LSL Rd, Ra, #imm4 — eg. LSL R5, R3, #7

Operation

$$
\begin{aligned}
Rd &\leftarrow Ra << \#imm4 \\
N &\leftarrow \text{if (Result} < 0\text{) then 1, else 0} \\
Z &\leftarrow \text{if (Result} = 0\text{) then 1, else 0} \\
V &\leftarrow \text{UNPREDICTABLE} \\
C &\leftarrow \text{UNPREDICTABLE}
\end{aligned}
$$

Description

The 16-bit word in GPR[Ra] is shifted left by the 4-bit amount specified in the instruction, shifting in zeros, and the result is placed into GPR[Rd].

Addressing Mode: Register-Immediate.

4.22 LSR — Logical Shift Right

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	Rd			Ra			0	imm4			

Syntax

LSR Rd, Ra, #imm4 eg. LSR R5, R3, #7

Operation

$$\begin{aligned}
\text{Rd} &\leftarrow \text{Ra} >> \#\text{imm4} \\
\text{N} &\leftarrow \text{if (Result} < 0\text{) then 1, else 0} \\
\text{Z} &\leftarrow \text{if (Result} = 0\text{) then 1, else 0} \\
\text{V} &\leftarrow \text{UNPREDICTABLE} \\
\text{C} &\leftarrow \text{UNPREDICTABLE}
\end{aligned}$$

Description

The 16-bit word in GPR[Ra] is shifted right by the 4-bit amount specified in the instruction, shifting in zeros, and the result is placed into GPR[Rd].

Addressing Mode: Register-Immediate.

4.23 ASR — Arithmetic Shift Right

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	Rd			Ra			0	imm4			

Syntax

ASR Rd, Ra, #imm4 — eg. ASR R5, R3, #7

Operation

Rd ← Ra >>> #imm4

N ← if (Result $<$ 0) then 1, else 0

Z ← if (Result = 0) then 1, else 0

V ← UNPREDICTABLE

C ← UNPREDICTABLE

Description

The 16-bit word in GPR[Ra] is shifted right by the 4-bit amount specified in the instruction, shifting in the sign bit of Ra, and the result is placed into GPR[Rd].

Addressing Mode: Register-Immediate.

4.24 LDW — Load Word

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	Rd			Ra			imm5				

Syntax

LDW Rd, [Ra, #imm5] eg. LDW R5, [R3, #7]

Operation

$$Rd \leftarrow Mem[Ra + \#imm5]$$
$$N \leftarrow N$$
$$Z \leftarrow Z$$
$$V \leftarrow V$$
$$C \leftarrow C$$

Description

Data is loaded from memory at the resultant address from addition of GPR[Ra] and the 5-bit immediate value specified in the instruction, and the result is placed into GPR[Rd].

Addressing Mode: Base Plus Offset.

4.25 STW — Store Word

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	Rd			Ra			imm5				

Syntax

STW Rd, [Ra, #imm5] eg. STW R5, [R3, #7]

Operation

$$\text{Mem}[\text{Ra} + \#\text{imm5}] \leftarrow \text{Rd}$$

$$N \leftarrow N$$

$$Z \leftarrow Z$$

$$V \leftarrow V$$

$$C \leftarrow C$$

Description

Data in GPR[Rd] is stored to memory at the resultant address from addition of GPR[Ra] and the 5-bit immediate value specified in the instruction.

Addressing Mode: Base Plus Offset.

4.26 LUI — Load Upper Immediate

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	Rd			imm8							

Syntax

LUI Rd #imm8 — eg. LUI R5, #93

Operation

$$\begin{aligned} Rd &\leftarrow \{\#imm8, 0\} \\ N &\leftarrow N \\ Z &\leftarrow Z \\ V &\leftarrow V \\ C &\leftarrow C \end{aligned}$$

Description

The 8-bit immediate value provided in the instruction is loaded into the top half in GPR[Rd], setting the bottom half to zero.

Addressing Mode: Register-Immediate.

4.27 LLI — Load Lower Immediate

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	Rd			imm8							

Syntax

LLI Rd #imm8 — eg. LLI R5, #93

Operation

$$Rd \leftarrow \{Rd[15:8], \#imm8\}$$

$$N \leftarrow N$$

$$Z \leftarrow Z$$

$$V \leftarrow V$$

$$C \leftarrow C$$

Description

The 8-bit immediate value provided in the instruction is loaded into the bottom half in GPR[Rd], leaving the top half unchanged.

Addressing Mode: Register-Immediate.

4.28 BR — Branch Always

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	imm8							

Syntax

BR LABEL — eg. BR .loop

Operation

$$PC \leftarrow PC + \#imm8$$

$$N \leftarrow N$$

$$Z \leftarrow Z$$

$$V \leftarrow V$$

$$C \leftarrow C$$

Description

Unconditionally branch to the resultant address from addition of PC and the 8-bit immediate value specified in the instruction. LABEL can be both a symbolic name or a numeric value, and is capable of jumping forwards or backwards.

Addressing Mode: PC Relative.

4.29 BNE — Branch If Not Equal

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	0	imm8							

Syntax

BNE LABEL — eg. BNE .loop

Operation

if (z=0) PC $\leftarrow$ PC + #imm8

N $\leftarrow$ N

Z $\leftarrow$ Z

V $\leftarrow$ V

C $\leftarrow$ C

Description

Conditionally branch to the resultant address from addition of PC and the 8-bit immediate value specified in the instruction if zero status flag (Z) equals zero. LABEL can be both a symbolic name or a numeric value, and is capable of jumping forwards or backwards.

Addressing Mode: PC Relative.

4.30 BE — Branch If Equal

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	1	imm8							

Syntax

BE LABEL — eg. BE .loop

Operation

if (z=1) PC $\leftarrow$ PC + #imm8

N $\leftarrow$ N

Z $\leftarrow$ Z

V $\leftarrow$ V

C $\leftarrow$ C

Description

Conditionally branch to the resultant address from addition of PC and the 8-bit immediate value specified in the instruction if zero status flag (Z) equals one. LABEL can be both a symbolic name or a numeric value, and is capable of jumping forwards or backwards.

Addressing Mode: PC Relative.

4.31 BLT — Branch If Less Than

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	imm8							

Syntax

BLT LABEL — eg. BLT .loop

Operation

if (n&!v OR !n&v) PC $\leftarrow$ PC + #imm8

N $\leftarrow$ N

Z $\leftarrow$ Z

V $\leftarrow$ V

C $\leftarrow$ C

Description

Conditionally branch to the resultant address from addition of PC and the 8-bit immediate value specified in the instruction if negative status flag and overflow status flag are not equivalent. LABEL can be both a symbolic name or a numeric value, and is capable of jumping forwards or backwards.

Addressing Mode: PC Relative.

4.32 BGE — Branch If Greater Than Or Equal

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	imm8							

Syntax

BGE LABEL — eg. BGE .loop

Operation

if (n&v OR !n&!v) PC $\leftarrow$ PC + #imm8

N $\leftarrow$ N

Z $\leftarrow$ Z

V $\leftarrow$ V

C $\leftarrow$ C

Description

Conditionally branch to the resultant address from addition of PC and the 8-bit immediate value specified in the instruction if negative status flag and overflow status flag are equivalent. LABEL can be both a symbolic name or a numeric value, and is capable of jumping forwards or backwards.

Addressing Mode: PC Relative.

4.33 BWL — Branch With Link

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	imm8							

Syntax

BWL LABEL   eg. BWL .loop

Operation

LR $\leftarrow$ PC + 1; PC $\leftarrow$ PC + #imm8

N $\leftarrow$ N

Z $\leftarrow$ Z

V $\leftarrow$ V

C $\leftarrow$ C

Description

Save the current program counter (PC) value plus one to the link register. Then unconditionally branch to the resultant address from addition of PC and the 8-bit immediate value specified in the instruction. LABEL can be both a symbolic name or a numeric value, and is capable of jumping forwards or backwards.

Addressing Mode: PC Relative.

4.34 RET — Return

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	imm8							

Syntax

RET — eg. RET

Operation

$$PC \leftarrow LR$$
$$N \leftarrow N$$
$$Z \leftarrow Z$$
$$V \leftarrow V$$
$$C \leftarrow C$$

Description

Unconditionally branch to the address stored in the link register (LR).

Addressing Mode: Register-Indirect.

4.35 JMP **Jump**

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	1	imm8							

Syntax

JMP Ra, #imm5 eg. JMP R3, #7

Operation

$$PC \leftarrow Ra + \#imm5$$

$$N \leftarrow N$$

$$Z \leftarrow Z$$

$$V \leftarrow V$$

$$C \leftarrow C$$

Description

Unconditionally jump to the resultant address from the addition of GPR[Ra] and the 5-bit immediate value specified in the instruction.

Addressing Mode: Base Plus Offset.

4.36 PUSH — Push From Stack

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	L	X	X	Ra			0	0	0	0	1

Syntax

PUSH Ra — eg. PUSH R3
PUSH LR — eg. PUSH LR

Operation

Mem[R7] $\leftarrow$ reg; R7 $\leftarrow$ R7 - 1

N $\leftarrow$ N

Z $\leftarrow$ Z

V $\leftarrow$ V

C $\leftarrow$ C

Description

'reg' corresponds to either a GPR or the link register, the contents of which are stored to the stack using the address stored in the stack pointer (R7). Then Decrement the stack pointer by one.

Addressing Modes: Register-Indirect, Postdecrement.

4.37 POP — Pop From Stack

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	L	X	X	Ra			0	0	0	0	1

Syntax

POP Ra — eg. POP R3
POP LR — eg. POP LR

Operation

$R7 \leftarrow R7 + 1$; $Mem[R7] \leftarrow reg$;

$N \leftarrow N$

$Z \leftarrow Z$

$V \leftarrow V$

$C \leftarrow C$

Description

Increment the stack pointer by one. Then 'reg' corresponds to either a GPR or the link register, the contents of which are retrieved from the stack using the address stored in the stack pointer (R7).

Addressing Modes: Register-Indirect, Preincrement.

4.38 RETI — Return From Interrupt

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	0	0	0	1	1	1	X	X	X	X	X

Syntax

RETI — eg. RETI

Operation

$$PC \leftarrow Mem[R7]$$
$$N \leftarrow N$$
$$Z \leftarrow Z$$
$$V \leftarrow V$$
$$C \leftarrow C$$

Description

Restore program counter to its value before interrupt occured, which is stored on the stack, pointed to be the stack pointer (R7). This must be the last instruction in an interrupt service routine.

Addressing Mode: Register-Indirect.

4.39 ENAI — Enable Interrupts

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	0	0	1	1	1	1	X	X	X	X	X

Syntax

ENAI — eg. ENAI

Operation

Set Interrupt Enable Flag

$N \leftarrow N$

$Z \leftarrow Z$

$V \leftarrow V$

$C \leftarrow C$

Description

Turn on interrupts by setting interrupt enable flag to true (1).

4.40 DISI — Disable Interrupts

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	0	1	0	1	1	1	X	X	X	X	X

Syntax

DISI eg. DISI

Operation

Reset Interrupt Enable Flag

$N \leftarrow N$

$Z \leftarrow Z$

$V \leftarrow V$

$C \leftarrow C$

Description

Turn off interrupts by setting interrupt enable flag to false (0).

4.41 STF — Store Status Flags

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	0	1	1	1	1	1	X	X	X	X	X

Syntax

STF — eg. STF

Operation

Mem[R7] $\leftarrow$ {12-bit 0, Z, C, V, N}; R7 $\leftarrow$ R7 - 1;

N $\leftarrow$ N

Z $\leftarrow$ Z

V $\leftarrow$ V

C $\leftarrow$ C

Description

Store contents of status flags to stack using address held in stack pointer (R7). Then decrement the stack pointer (R7) by one.

Addressing Modes: Register-Indirect, Postdecrement.

4.42 LDF — Load Status Flags

Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	0	0	1	1	1	X	X	X	X	X

Syntax

LDF — eg. LDF

Operation

$$
\begin{aligned}
R7 &\leftarrow R7 + 1 \\
N &\leftarrow \text{Mem}[R7][0] \\
Z &\leftarrow \text{Mem}[R7][3] \\
V &\leftarrow \text{Mem}[R7][1] \\
C &\leftarrow \text{Mem}[R7][2]
\end{aligned}
$$

Description

Increment the stack pointer (R7) by one. Then load content of status flags with lower 4 bits of value retrieved from stack using address held in stack pointer (R7).

Addressing Modes: Register-Indirect, Preincrement.

5 Programming Tips

Lorem Ipsum. . .

6 Assembler

The current instruction set architecture includes an assembler for converting assembly language into hexadecimal. This chapter outlines the required formatting and available features of this assembler.

6.1 Instruction Formatting

Each instruction must be formatted using the following syntax. Here "[...]" indicates an optional field:

```
[.LABELNAME] MNEMONIC, OPERANDS, ..., :[COMMENTS]
```

For example:

```
.loop ADDI, R5, R3, #5 :Add 5 to R3
```

Comments may be added by preceding them with either : or ;

Accepted general purpose register values are: R0, R1, R2, R3, R4, R5, R6, R7, SP. These can be upper or lower case and SP is equivalently evaluated to R7.

Branch instructions take a symbolic reference to the destination. Each type of branch supports moving up to 127 lines forward, or 128 lines backwards. But if a branch is over this limitation, the assembler will automatically create additional instructions to enable greater distances. Each additional branch added will cause two more lines of code to be added to the outputted file.

All label names must begin with a '.' while `.ISR`/`.isr` and `.define` are special cases used for the interrupt service routine and variable definitions respectively.

Instruction-less or comments only lines are allowed within the assembly file.

Special Case Label
The .ISR/.isr label is reserved for the interrupt service routine and may be located anywhere within the file but must finish with a 'RETI' instruction. Branches may occur within the ISR, but are not allowed into this service routine with the exception of a return from a separate subroutine.

6.2 Assembler Directives

Symbolic label names are supported for branch-type instructions. Following the previous syntax definition for '.LABELNAME', they can be used instead of numeric branching provided they branch no further than the maximum distance allowed for the instruction used. Definitions are supported by the assembler. They are used to assign meaningful names to the GPRs to aid with programming. Definitions can occur at any point within the file and create a mapping from that point onwards. Different names can be assigned to the same register, but only one is valid at a time.

The accepted syntax for definitions is:

```
.define NAME REGISTER
```

6.3 Running The Assembler

The assembler is a python executable and is run by typing "./assemble.py". Alternatively, the assembler can be placed in a folder on the users path and executed by running "assemble.py". It supports Python versions 2.4.3 to 2.7.3. A help prompt is given by the script if the usage is not correct, or given a `-h` or `--help` argument.

By default, the script will output the assembled hex to a file with the same name, but with a '.hex' extension in the same directory. The user can specify a different file to use by using a `-o filename.hex` or `--output=filename.hex` argument to the script. The output file can also be a relative or absolute path to a different directory.

The full usage for the script is seen in listing 1. This includes the basic rules for writing the assembly language and a version log.

Listing 1: Assembler help prompt

```
$> assemble.py
```

```
Usage: assemble.py [-o outfile] input

---Team R4 Assembler Help---
-------Version:
 1 (CMPI addition onwards)
 2 (Changed to final ISA, added special case I's and error checking
 3 (Ajr changes - Hex output added, bug fix)
 4 (Added SP symbol)
 5 (NOP support added, help added) UNTESTED
 6 (Interrupt support added [ENAI, DISI, RETI])
 7 (Checks for duplicate Labels)
 8 (Support for any ISR location & automated startup code entry)
 9 (Support for .define)
 10 (Changed usage)
 Current is most recent iteration
Commenting uses : or ;
Labels start with '.': SPECIAL .ISR/.isr -> Interrupt Service Routine)
                       SPECIAL .define -> define new name for General Purpose Register, .define NAME R0-R7/SP
Instruction Syntax: .[LABELNAME] MNEUMONIC, OPERANDS, ..., :[COMMENTS]
Registers: R0, R1, R2, R3, R4, R5, R6, R7==SP
Branching: Symbolic and Numeric supported

Notes:
 Input files are assumed to end with a .asm extension
 Immediate value sizes are checked
 Instruction-less lines allowed
 .ISR may be located anywhere in file
 .define may be located anywhere, definition valid from location in file onwards, may replace existing definitions


Options:
  -h, --help            show this help message and exit
  -o FILE, --output=FILE
                        output file for the assembled output
```

6.4 Error Messages

Code	Description
ERROR1	Instruction mneumonic is not recognized
ERROR2	Register code within instruction is not recognized
ERROR3	Branch condition code is not recognised
ERROR4	Attempting to branch to undefined location
ERROR5	Instruction mneumonic is not recognized
ERROR6	Attempting to shift by more than 16 or perform a negative shift
ERROR7	Magnitude of immediate value for ADDI, ADCI, SUBI, SUCI, LDW or STW is too large
ERROR8	Magnitude of immediate value for CMPI or JMP is too large
ERROR9	Magnitude of immediate value for ADDIB, SUBIB, LUI or LLI is too large
ERROR10	Attempting to jump more than 127 forward or 128 backwards
ERROR11	Duplicate symbolic link names
ERROR12	Illegal branch to ISR
ERROR13	Multiple ISRs in file
ERROR14	Invalid formatting for .define directive

7 Programs

Every example program in this section uses R7 as a stack pointer which is initialised to the by the program to 0x07D0.The simulation environment contains an area of an area of memory with 2048 locations and memory mapped deices. There are 16 switches at location 0x0800, 16 LEDs at location 0x0801 and a serial io device which can be read from location 0xA000 and has a control register at location 0xA001.

7.1 Multiply

The code for the multiply program is held in Appendix A.1 listing 7. A sixteen bit number is read from input switches, split in to lower and upper bytes which are then multiplied. The resulting sixteen bit word is written to the LEDs before reaching a terminating loop. Equation (1) formally describes the algorithm disregarding limitations.

$$A = M \times Q = \sum_{i=0}^{\infty} 2^i M_i Q \ where \ M_i \in \{0, 1\} \tag{1}$$

The subroutine operation is described in listing 2, using C. If the result is greater than or equal to 2^{16} the subroutine will fail and return zero. The lowest bit of the multiplier controls the accumulator and the overflow check. The multiplier is shifted right and the quotient is shifted left at every iteration. An unconditional branch is used to keep the algorithm in a while loop. The state of the multiplier is compared at every iteration against zero when the algorithm is finished. As size of the multiplier controls the number of iterations a comparison is made on entry to use the smallest operand.

Listing 2: Multiply Subroutine

```
uint16_t multi(uint16_t op1, op2){
    uint16_t A,M,Q;
    A = 0;
    if(op1 < op2){              // Make M small, less loops
        M = op1; Q = op2;
    }else{
        M = op2; Q = op1;
    }
    while(1){                   // No loop counter
        if(M & 0x0001){         // LSb
```

```
                A = A + Q;
                if(A > 0xFFFF){       // Using carry flag
                    return 0;         // Overflow - fail
                }
            }
            M = M >> 1;
            if(0 == M){
                return A;             // Finished - pass
            }
            if(Q & 0x8000){
                return 0;             // Q >= 2^16 - fail
            }
            Q = Q << 1;
        }
}
```

7.2 Factorial

The code for the factorial program is held in Appendix A.2 listing 8. It is possible to calculate the factorial of any integer value between 0 and 8 inclusive. The subroutine is called which in turn calls the multiply subroutine discussed in section 7.1. The factorial subroutine does no parameter checking but the multiply code does so if overflow does occur zero is propagated and returned; zero is not a possible factorial. The result is calculated recursively as described using C in listing 3. Large values can cause stack overflow the main body of code makes sure inputs, read from the switches, are sufficiently small.

Listing 3: Recursive Factorial Subroutine

```
uint16_t fact(uint16_t x){
    if(x == 0){
        return 1;                    // 0! = 1
    }
    return multi(x,fact(x-1));    // Recurrsive
}
```

7.3 Random

The code for the random program is held in Appendix A.3 listing 9. A random series of numbers is achieved by simulating the 16 bit linear feedback shift register in Figure 1. This produces a new number every 16 sixteen clock cycles so in this case a simulation subroutine is called 16 times. A seed taken from switches and passed to the first subroutine call via the stack is altered and passed to the next subroutine call. No more stack operations are performed. A load from the stack pointer is used write a new random number to LEDs. All contained within an unconditional branch but a loop counter is used control write and reset.

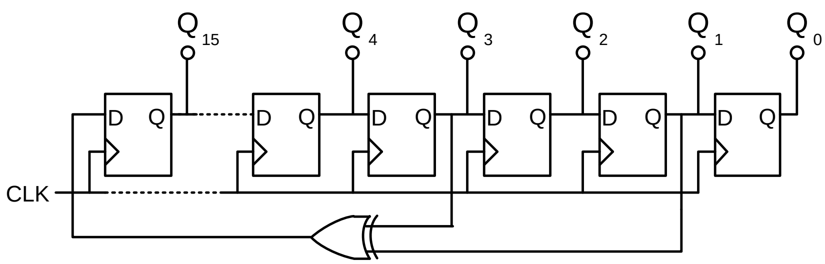

Figure 1: 16 Bit Linear Feedback Shift Register.

A two input XOR gate is simulated using the XOR operation along with shifting to compare bits in different locations. Bits 2 and 4 are used as inputs so a logical shift left by two is used to align them at the bit 4 position. Masking the output value is used feedback to the top bit. This is described using C in listing 4.

Listing 4: Linear Feedback Shift Register Subroutine

```
uint16_t rand(uint16_t last){
    uint16_t next,test;
    next = last >> 1;              // The shift
    test = last << 2;              // Compare different bits
    test = test ^ last;
    if(test & 0x0008){             // Feedback to top
        return (next | 0x8000);
    }
    return next;
}
```

7.4 Interrupt

The code for the interrupt program is held in Appendix A.4 listing 10. This is the most complex example and makes use of both the multiply and factorial subroutines in sections 7.1 and 7.2 respectively. The interrupt services the serial device by writing data to a 4 byte circular buffer. A main program check to see if data is in the buffer then and if so calculates the factorial writing the result to the LEDs. The buffer is purposefully small to test overflow.

Listing 5: Serial Device Interrupt Service Request

```
#define TOP     0x0206
#define BOTTOM  0x0202
#define WRITE   0x0201
#define READ    0x0200
#define SERIAL  0xA000

isr(){
    uint16_t data,readPtri,writePtr;
    asm("DISI");                      // critical op
    data = read(SERIAL);
    asm("ENAI");                      // nested ints
    readPtr = read(READ);
    writePtr = read(WRITE);
    if(((readPtr-1) == writePtr)    ||
        (readPtr == BOTTOM)         ||
        (writePtr == (TOP-1))       ){
        asm("RETI");                  // full, don't write
    }
    if(readPtr == BOTTOM)
    write(readPtr,data);              // write to buffer
    writePtr++;
    if(writePtr == TOP){
        writePtr = BOTTOM;
    }else{
        writePtr++;
    }
    write(WRITE,writePtr);
    asm("RETI")
}

void main(){
    uint16_t readPtr,writePtri,data;
    do{
        readPtr = read(READ);
        writePtr = read(WRITE);
    }while(readPtr == writePtr)
    data = read(readPtr)
```

```

        fact();
}
```

8 Simulation

8.1 Running the simulations

A register window could also be done for this section too

Sim.py needs a fair bit of change. If we have time, this could be altered to use Iains dir structure. This section highlights how to use his script and our assembler.

Before the simulator is invoked, the assembler should be run. This is discussed in section 6.3. It can be done from within the programs directory (`/design/fcde/verilog/programs`) by running, for example, `assemble.py multiply`

The script "simulate" is an executable shell script. It is run from the terminal in the directory `/design/fcde/verilog`. This supports running simulations of a full verilog model, cross simulation and a fully extracted simulation. Usage is as follows:

```
./simulate type program [definitions]
```

The 'type' can be one of the following: *behavioural, mixed, extracted.* 'Program' is a relative path to the assembled hex file, usually located in the programs folder. Extra definitions can also be included to set the switch value or serial data input.

The serial data file used is located in the programs directory. This is a hex file with white space separated values of the form " time data". The data is then sent at the time to the processor by the serial module. An example serial data hex file is shown in listing 6.

Listing 6: Example serial data file

```
// Hex file to specify serial data input
//
//
  248   7
  48F   6
  6D6   5
  91D   4
  B64   7
  DAB   5
```

```
36B1    3
6D61    2
```

Below is a complete list of commands to run all programs on all versions of the processor. ‘Number’ is a user defined decimal value to set the switches.

- ./assembler/assemble.py programs/multiply.asm
  ./simulate behavioural programs/multiply.hex +define+switch_value=number

- ./assembler/assemble.py programs/multiply.asm
  ./simulate mixed programs/multiply.hex +define+switch_value=number

- ./assembler/assemble.py programs/multiply.asm
  ./simulate extracted programs/multiply.hex +define+switch_value=number

- ./assembler/assemble.py programs/random.asm
  ./simulate behavioural programs/random.hex +define+switch_value=number

- ./assembler/assemble.py programs/random.asm
  ./simulate mixed programs/random.hex +define+switch_value=number

- ./assembler/assemble.py programs/random.asm
  ./simulate extracted programs/random.hex +define+switch_value=number

- ./assembler/assemble.py programs/factorial.asm
  ./simulate behavioural programs/factorial.hex +define+switch_value=number

- ./assembler/assemble.py programs/factorial.asm
  ./simulate mixed programs/factorial.hex +define+switch_value=number

- ./assembler/assemble.py programs/factorial.asm
  ./simulate extracted programs/factorial.hex +define+switch_value=number

- ./assembler/assemble.py programs/interrupt.asm
  ./simulate behavioural programs/interrupt.hex programs/serial_data.hex

- ./assembler/assemble.py programs/interrupt.asm
  ./simulate mixed programs/interrupt.hex programs/serial_data.hex

Table 1: Clock cycles required for each program to run

Make these more accurate when AJR has finished playing around

Program	Clock Cycles
Multiply	900
Factorial	6000
Random	
Interrupt	30000

- `./assembler/assemble.py programs/interrupt.asm`
  `./simulate extracted programs/interrupt.hex programs/serial_data.hex`

A scan path simulation can also be run. This is done by running `ncverilog -sv +gui +ncaccess+r stimulus.sv opcodes.svh cpu.sv` for a GUI or `ncverilog -sv stimulus.sv opcodes.svh cpu.sv -exit` for a command line simulation. This test pulses a signal on the SDI line, and verifies a pulse is seen on the output. The clock cycles, and therefore the number of registers, are counted and reported upon success of the simulation.

The number of clock cycles for each program to fully run is shown in table 1. Factorial run time is given for an input of 8 and is the worst case. Interrupt is dependant on the serial data input and the time is given for the serial data file mentioned above.

A dissembler is also implemented in System Verilog to aid debugging. It is an ASCII formatted array implemented at the top level of the simulation. It is capable of reading the instruction register with in the design, and reconstructing the assembly language of the instruction and is supported in behavioural, mixed and extracted simulations. It will show the opcode, register addresses and immediate values. It is automatically included by the TCL script. The TCL script also opens a waveform window and adds important signals.

A Code Listings

All code listed in this section is passed to the assembler *as is* and has been verfied using the final design of the processor.

A.1 Multiply

Listing 7: multiply.asm

```
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        LUI     SP, #7        ; Init SP
        LLI     SP, #208
        LUI     R3, #8        ; SWs addr
        LLI     R3, #0
        LDW     R0,[R3,#0]    ; READ SWs
        LUI     R1, #0
        LLI     R1, #255      ; 0x00FF in R1
        AND     R1,R0,R1      ; Lower byte SWs in R1
        LSR     R0,R0,#8      ; Upper byte SWs in R0
        PUSH    R0            ; Op1
        PUSH    R1            ; Op2
        SUB     R2,R2,R2      ; Zero required
        PUSH    R2            ; Place holder is zero
        BWL     .multi        ; Run Subroutine
        POP     R1            ; Result
        ADDIB   SP,#2         ; Duummy pop
        ADDIB   R3,#1         ; Address of LEDS
```

```
        STW     R1,[R3,#0]   ; Result on LEDS
.end    BR      .end         ; Finish loop
.multi  PUSH    R0
        PUSH    R1
        PUSH    R2
        PUSH    R3
        PUSH    R4
        PUSH    R5
        PUSH    R6
        LDW     R0,[SP,#8]   ; R0 - Multiplier
        LDW     R1,[SP,#9]   ; R1 - Quotient
        CMP     R0,R1
        BLT     .nSw         ; Branch if M < Q
        ADDI    R2,R1,#0     ; Make M the smallest
        ADDI    R1,R0,#0
        ADDI    R0,R2,#0
.nSw    SUB     R2,R2,R2     ; R2 - Accumulator
        ADDI    R3,R2,#1     ; R3 - 0x0001
        LUI     R4,#128      ; R4 - 0x8000
        LLI     R4,#0
.mloop  AND     R6,R0,R3     ; R6 - Cmp var
        CMPI    R6,#1
        BNE     .nAcc
        SUB     R3,R3,R3
        ADD     R2,R2,R1     ; A = A + Q
        ADCI    R3,R3,#1
        CMPI    R3,#2
        BE      .fail        ; OV
.nAcc   LSR     R0,R0,#1     ; M = M >> 1
        CMPI    R0,#0
        BE      .done
        AND     R5,R4,R1
        CMPI    R5,#0
        BNE     .fail
        LSL     R1,R1,#1     ; Q = Q << 1
        BR      .mloop
.done   STW     R2,[SP,#7]   ; Res on stack frame
        POP     R6
        POP     R5
        POP     R4
        POP     R3
        POP     R2
        POP     R1
        POP     R0
        RET
```

```
.fail   SUB     R2,R2,R2      ; OV - ret 0
        BR      .done
```

A.2 Factorial

Listing 8: factorial.asm

```
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        ADDIB R0,#0
        LUI     R7, #7
        LLI     R7, #208
        LUI     R0, #8        ; Address in R0
        LLI     R0, #0
        LDW     R1,[R0,#0]    ; Read switches into R1
        PUSH    R1            ; Pass para
        BWL     .fact         ; Run Subroutine
        POP     R3            ; Para overwritten with result
        ADDIB   R0,#1
        STW     R3,[R0,#0]    ; Result on LEDS
.end    BR      .end          ;  finish loop
.fact   PUSH    R0
        PUSH    R1
        PUSH    LR
        LDW     R1,[SP,#3]    ; Get para
        ADDIB   R1,#0
        BE      .retOne           ; 0! = 1
        SUBI    R0,R1,#1
        PUSH    R0            ; Pass para
```

```
        BWL     .fact       ; The output remains on the stack
        PUSH    R1          ; Pass para
        SUBIB   SP,#1       ; Placeholder
        BWL     .multi
        POP     R1          ; Get res
        ADDIB   SP,#2       ; pop x 2
        STW     R1,[SP,#3]
        POP     LR
        POP     R1
        POP     R0
        RET
.retOne ADDIB   R1,#1       ; Avoid jump checking
        STW     R1,[SP,#3]
        POP     LR
        POP     R1
        POP     R0
        RET
.multi  PUSH    R0
        PUSH    R1
        PUSH    R2
        PUSH    R3
        PUSH    R4
        PUSH    R5
        PUSH    R6
        LDW     R0,[SP,#8]  ; R0 - Multiplier
        LDW     R1,[SP,#9]  ; R1 - Quotient
        CMP     R0,R1
        BLT     .nSw        ; Branch if M < Q
        ADDI    R2,R1,#0    ; Make M the smallest
        ADDI    R1,R0,#0
        ADDI    R0,R2,#0
.nSw    SUB     R2,R2,R2    ; R2 - Accumulator
        ADDI    R3,R2,#1    ; R3 - 0x0001
        LUI     R4,#128     ; R4 - 0x8000
        LLI     R4,#0
.mloop  AND     R6,R0,R3    ; R6 - Cmp var
        CMPI    R6,#1
        BNE     .nAcc
        SUB     R3,R3,R3
        ADD     R2,R2,R1    ; A = A + Q
        ADCI    R3,R3,#1
        CMPI    R3,#2
        BE      .fail       ; OV
.nAcc   LSR     R0,R0,#1    ; M = M >> 1
        CMPI    R0,#0
```

```
            BE      .done
            AND     R5,R4,R1
            CMPI    R5,#0
            BNE     .fail
            LSL     R1,R1,#1     ; Q = Q << 1
            BR      .mloop
.done       STW     R2,[SP,#7]   ; Res on stack frame
            POP     R6
            POP     R5
            POP     R4
            POP     R3
            POP     R2
            POP     R1
            POP     R0
            RET
.fail       SUB     R2,R2,R2     ; OV - ret 0
            BR      .done
```

A.3 Random

Listing 9: random.asm

```
            ADDIB   R0,#0
ADDIB       R0,#0
ADDIB       R0,#0
ADDIB       R0,#0
ADDIB       R0,#0
ADDIB       R0,#0
ADDIB       R0,#0
ADDIB       R0,#0
ADDIB       R0,#0
ADDIB       R0,#0
ADDIB       R0,#0
ADDIB       R0,#0
ADDIB       R0,#0
ADDIB       R0,#0
ADDIB       R0,#0
ADDIB       R0,#0
ADDIB       R0,#0
ADDIB       R0,#0
ADDIB       R0,#0
            LUI     SP,#7        ; Init SP
            LLI     SP,#208
            LUI     R0,#8        ; SW Address in R0
```

```
            LLI     R0,#0
            LDW     R1,[R0,#0]   ; Read switches into R1
            ADDIB   R0,#1        ; Address of LEDS in R0
            PUSH    R1
.reset      SUB     R4,R4,R4     ; Reset Loop counter
.loop       BWL     .rand
            CMPI    R4,#15
            BE      .write
            ADDIB   R4,#1        ; INC loop counter
            BR      .loop
.write      LDW     R1,[SP,#0]   ; No pop as re-run
            STW     R1,[R0,#0]   ; Result on LEDS
            BR      .reset
.rand       PUSH    R0           ; LFSR Sim
            PUSH    R1           ; Protect regs
            PUSH    R2
            LDW     R0,[SP,#3]   ; Last reg value
            LSL     R1,R0,#2     ; Shift Bit 4 <- 2
            XOR     R1,R0,R1     ; xor Gate
            LSR     R0,R0,#1     ; Shifted reg
            LUI     R2,#0
            LLI     R2,#8
            AND     R1,R2,R1     ; Mask off Bit 4
            CMPI    R1,#0
            BNE     .done
            LUI     R1,#128
            LLI     R1,#0
            OR      R0,R0,R1     ; or with 0x8000
.done       STW     R0,[SP,#3]
            POP     R2
            POP     R1
            POP     R0
            RET
```

A.4 Interrupt

Listing 10: interrupt.asm

```
            ADDIB R0,#0
            ADDIB R0,#0
            ADDIB R0,#0
            ADDIB R0,#0
            ADDIB R0,#0
            ADDIB R0,#0
```

```
          ADDIB R0,#0
          ADDIB R0,#0
          ADDIB R0,#0
          ADDIB R0,#0
          ADDIB R0,#0
          ADDIB R0,#0
          ADDIB R0,#0
          ADDIB R0,#0
          ADDIB R0,#0
          ADDIB R0,#0
          ADDIB R0,#0
          ADDIB R0,#0
          DISI                  ; Reset is off anyway
          LUI    R7, #7
          LLI    R7, #208
          LUI    R0, #2         ; R0 is read ptr    0x0200
          LLI    R0, #0
          ADDI   R1,R0,#2       ; 0x0202
          STW    R1,[R0,#0]     ; Read ptr set to   0x0202
          STW    R1,[R0,#1]     ; Write ptr set to  0x0202
          LUI    R0,#160        ; Address of Serial control reg
          LLI    R0,#1
          LUI    R1,#0
          LLI    R1,#1          ; Data to enable ints
          STW    R1,[R0,#0]     ; Store 0x001 @ 0xA001
          ENAI
          BR     .main
.isr      STF                   ; Keep flags, disable auto
          PUSH   R0             ; Save only this for now
          LUI    R0,#160
          LLI    R0,#0
          LDW    R0,[R0,#0]     ; R1 contains read serial data
          ENAI                  ; Don't miss event
          PUSH   R1
          PUSH   R2
          PUSH   R3
          PUSH   R4
          LUI    R1,#2
          LLI    R1,#0
          LDW    R2,[R1,#0]     ; R2 contains read ptr
          ADDI   R3,R1,#1
          LDW    R4,[R3,#0]     ; R4 contain the write ptr
          SUBIB  R2,#1          ; Get out if W == R - 1
          CMP    R4,R2
          BE     .isrOut
```

```
        ADDIB   R2,#1
        LUI     R1,#2
        LLI     R1,#2
        CMP     R2,R1
        BNE     .write
        ADDIB   R1,#3
        CMP     R4,R1
        BE      .isrOut
.write  STW     R0,[R4,#0]    ; Write to buffer
        ADDIB   R4,#1
        LUI     R1,#2
        LLI     R1,#6
        CMP     R1,R4
        BNE     .wrapW
        SUBIB   R4,#4
.wrapW  STW     R4,[R3,#0]    ; Inc write ptr
.isrOut POP     R4
        POP     R3
        POP     R2
        POP     R1
        POP     R0
        LDF
        RETI
.main   LUI     R0, #2        ; Read ptr address in R0
        LLI     R0, #0
        LDW     R2,[R0,#0]    ; Read ptr in R2
        LDW     R3,[R0,#1]    ; Write ptr in R3
        CMP     R2,R3
        BE      .main         ; Jump back if the same
        LDW     R3,[R2,#0]    ; Load data out of buffer
        ADDIB   R2,#1         ; Inc read ptr
        SUB     R0,R0,R0
        LUI     R0,#2
        LLI     R0,#6
        SUB     R0,R0,R2
        BNE     .wrapR
        SUBIB   R2,#4
.wrapR  LUI     R0, #2        ; Read ptr address in R0
        LLI     R0, #0
        STW     R2,[R0,#0]    ; Store new read pointer
        SUB     R4,R4,R4
        LLI     R4,#15
        AND     R3,R4,R3
        CMPI    R3,#8
        BE      .do
```

```
        LLI     R4,#7
        AND     R3,R3,R4
.do     PUSH    R3
        BWL     .fact
        POP     R3
        LUI     R4,#8
        LLI     R4,#1         ; Address of LEDs
        STW     R3,[R4,#0]    ; Put factorial on LEDs
        BR      .main             ; look again
.fact   PUSH    R0
        PUSH    R1
        PUSH    LR
        LDW     R1,[SP,#3]    ; Get para
        ADDIB   R1,#0
        BE      .retOne           ; 0! = 1
        SUBI    R0,R1,#1
        PUSH    R0            ; Pass para
        BWL     .fact         ; The output remains on the stack
        PUSH    R1            ; Pass para
        SUBIB   SP,#1         ; Placeholder
        BWL     .multi
        POP     R1            ; Get res
        ADDIB   SP,#2         ; pop x 2
        STW     R1,[SP,#3]
        POP     LR
        POP     R1
        POP     R0
        RET
.retOne ADDIB   R1,#1         ; Avoid jump checking
        STW     R1,[SP,#3]
        POP     LR
        POP     R1
        POP     R0
        RET
.multi  PUSH    R0
        PUSH    R1
        PUSH    R2
        PUSH    R3
        PUSH    R4
        PUSH    R5
        PUSH    R6
        LDW     R0,[SP,#8]    ; R0 - Multiplier
        LDW     R1,[SP,#9]    ; R1 - Quotient
        CMP     R0,R1
        BLT     .nSw          ; Branch if M < Q
```

```
        ADDI    R2,R1,#0      ; Make M the smallest
        ADDI    R1,R0,#0
        ADDI    R0,R2,#0
.nSw    SUB     R2,R2,R2      ; R2 - Accumulator
        ADDI    R3,R2,#1      ; R3 - 0x0001
        LUI     R4,#128       ; R4 - 0x8000
        LLI     R4,#0
.mloop  AND     R6,R0,R3      ; R6 - Cmp var
        CMPI    R6,#1
        BNE     .nAcc
        SUB     R3,R3,R3
        ADD     R2,R2,R1      ; A = A + Q
        ADCI    R3,R3,#1
        CMPI    R3,#2
        BE      .fail         ; OV
.nAcc   LSR     R0,R0,#1      ; M = M >> 1
        CMPI    R0,#0
        BE      .done
        AND     R5,R4,R1
        CMPI    R5,#0
        BNE     .fail
        LSL     R1,R1,#1      ; Q = Q << 1
        BR      .mloop
.done   STW     R2,[SP,#7]    ; Res on stack frame
        POP     R6
        POP     R5
        POP     R4
        POP     R3
        POP     R2
        POP     R1
        POP     R0
        RET
.fail   SUB     R2,R2,R2      ; OV - ret 0
        BR      .done
```