

ELEC6027 - VLSI Design Project : Programmers Guide

Team R4

28th April, 2014

Todo list

- A register window could also be done for this section too 58
- 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. 58
- Make these more accurate when AJR has finished playing around . . 61

Contents

1	Introduction	5
2	Architecture	5
3	Register Description	5
4	Instruction Set	7
4.1	General Instruction Formatting	8
4.2	ADD	10
4.3	ADDI	11
4.4	ADDIB	12
4.5	ADC	13
4.6	ADCI	14
4.7	NEG	15
4.8	SUB	16
4.9	SUBI	17
4.10	SUBIB	18
4.11	SUC	19
4.12	SUCI	20
4.13	CMP	21
4.14	CMPI	22
4.15	AND	23
4.16	OR	24
4.17	XOR	25
4.18	NOT	26
4.19	NAND	27
4.20	NOR	28
4.21	LSL	29
4.22	LSR	30
4.23	ASR	31
4.24	LDW	32
4.25	STW	33
4.26	LUI	34
4.27	LLI	35
4.28	BR	36
4.29	BNE	37

4.30	BE	38
4.31	BLT	39
4.32	BGE	40
4.33	BWL	41
4.34	RET	42
4.35	JMP	43
4.36	PUSH	44
4.37	POP	45
4.38	RETI	46
4.39	ENAI	47
4.40	DISI	48
4.41	STF	49
4.42	LDF	50
5	Programming Tips	51
6	Assembler	51
6.1	Instruction Formatting	51
6.2	Assembler Directives	52
6.3	Running The Assembler	52
6.4	Error Messages	54
7	Programs	54
7.1	Multiply	55
7.2	Factorial	56
7.3	Random	56
7.4	Interrupt	57
8	Simulation	58
8.1	Running the simulations	58
A	Code Listings	62
A.1	Multiply	62
A.2	Factorial	64
A.3	Random	66
A.4	Interrupt	67

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

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

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

$Rd \leftarrow Ra + Rb$

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

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

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

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

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.

4.3 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

$Rd \leftarrow Ra + \#imm5$

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

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

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

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

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

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

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

$Rd \leftarrow Rd + \#imm8$

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

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

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

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

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

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

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

$Rd \leftarrow Ra + Rb + C$

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

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

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

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

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

$Rd \leftarrow Ra + \#imm5 + C$

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

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

$V \leftarrow \text{if } (Ra > 0 \text{ and } (\#imm5 + CFlag) > 0 \text{ and } \text{Result} < 0) \text{ or}$
 $(Ra < 0 \text{ and } (\#imm5 + CFlag) < 0 \text{ and } \text{Result} > 0) \text{ then } 1, \text{ else } 0$

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

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

$Rd \leftarrow 0 - Ra$

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

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

$V \leftarrow 0$

$C \leftarrow 0$

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

$Rd \leftarrow Ra - Rb$

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

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

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

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

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

$Rd \leftarrow Ra - \#imm5$

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

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

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

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

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

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

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

$Rd \leftarrow Rd - \#imm8$

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

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

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

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

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

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

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

$Rd \leftarrow Ra - Rb - C$

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

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

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

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

Description

The 16-bit word in GPR[Rb] 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-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

$Rd \leftarrow Ra - \#imm5 - C$

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

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

$V \leftarrow \text{if } (Ra > 0 \text{ and } (\#imm5 - CFlag) > 0 \text{ and } Result < 0) \text{ or } (Ra < 0 \text{ and } (\#imm5 - CFlag) < 0 \text{ and } Result > 0) \text{ then } 1, \text{ else } 0$

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

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

Ra - Rb

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$Rd \leftarrow Ra \text{ AND } Rb$

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

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

$V \leftarrow \text{UNPREDICTABLE}$

$C \leftarrow \text{UNPREDICTABLE}$

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

$Rd \leftarrow Ra \text{ OR } Rb$

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

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

$V \leftarrow \text{UNPREDICTABLE}$

$C \leftarrow \text{UNPREDICTABLE}$

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

$Rd \leftarrow Ra \text{ XOR } Rb$

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

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

$V \leftarrow \text{UNPREDICTABLE}$

$C \leftarrow \text{UNPREDICTABLE}$

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

$Rd \leftarrow \text{NOT } Ra$

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

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

$V \leftarrow \text{UNPREDICTABLE}$

$C \leftarrow \text{UNPREDICTABLE}$

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

$Rd \leftarrow Ra \text{ NAND } Rb$

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

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

$V \leftarrow \text{UNPREDICTABLE}$

$C \leftarrow \text{UNPREDICTABLE}$

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

$Rd \leftarrow Ra \text{ NOR } Rb$

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

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

$V \leftarrow \text{UNPREDICTABLE}$

$C \leftarrow \text{UNPREDICTABLE}$

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

$Rd \leftarrow Ra \ll \#imm4$

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

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

$V \leftarrow \text{UNPREDICTABLE}$

$C \leftarrow \text{UNPREDICTABLE}$

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

$Rd \leftarrow Ra \gg \#imm4$

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

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

$V \leftarrow \text{UNPREDICTABLE}$

$C \leftarrow \text{UNPREDICTABLE}$

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

$Rd \leftarrow Ra \ggg \#imm4$

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

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

$V \leftarrow \text{UNPREDICTABLE}$

$C \leftarrow \text{UNPREDICTABLE}$

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

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

$N \leftarrow N$

$Z \leftarrow Z$

$V \leftarrow V$

$C \leftarrow C$

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

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

$\text{N} \leftarrow \text{N}$

$\text{Z} \leftarrow \text{Z}$

$\text{V} \leftarrow \text{V}$

$\text{C} \leftarrow \text{C}$

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

$Rd \leftarrow \{\#imm8, 0\}$

$N \leftarrow N$

$Z \leftarrow Z$

$V \leftarrow V$

$C \leftarrow C$

Description

The 8-bit immediate value provided in the instruction is loaded into the 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

$\text{Mem}[\text{R7}] \leftarrow \text{reg}; \text{R7} \leftarrow \text{R7} - 1$

$\text{N} \leftarrow \text{N}$

$\text{Z} \leftarrow \text{Z}$

$\text{V} \leftarrow \text{V}$

$\text{C} \leftarrow \text{C}$

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

$R7 \leftarrow R7 + 1$; $\text{Mem}[R7] \leftarrow \text{reg}$;

$N \leftarrow N$

$Z \leftarrow Z$

$V \leftarrow V$

$C \leftarrow C$

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

$PC \leftarrow \text{Mem}[R7]$

$N \leftarrow N$

$Z \leftarrow Z$

$V \leftarrow V$

$C \leftarrow C$

Description

Restore program counter to its value before interrupt occurred, 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

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

$N \leftarrow N$

$Z \leftarrow Z$

$V \leftarrow V$

$C \leftarrow C$

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

$R7 \leftarrow R7 + 1$

$N \leftarrow \text{Mem}[R7][0]$

$Z \leftarrow \text{Mem}[R7][3]$

$V \leftarrow \text{Mem}[R7][1]$

$C \leftarrow \text{Mem}[R7][2]$

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

```
1 $> assemble.py
```

```

2 Usage: assemble.py [-o outfile] input
3
4 —Team R4 Assembler Help—
5 —Version:
6   1 (CMPI addition onwards)
7   2 (Changed to final ISA, added special case I's and error
   checking
8   3 (Ajr changes – Hex output added, bug fix)
9   4 (Added SP symbol)
10  5 (NOP support added, help added) UNTESTED
11  6 (Interrupt support added [ENAI, DISI, RETI])
12  7 (Checks for duplicate Labels)
13  8 (Support for any ISR location & automated startup code entry)
14  9 (Support for .define)
15 10 (Changed usage)
16  Current is most recent iteration
17  Commenting uses : or ;
18  Labels start with '.': SPECIAL .ISR/.isr-> Interrupt Service
   Routine)
19
   SPECIAL .define -> define new name for
   General Purpose Register, .define NAME R0-R7/SP
20 Instruction Syntax: .[LABELNAME] MNEUMONIC, OPERANDS, ..., :[
   COMMENTS]
21 Registers: R0, R1, R2, R3, R4, R5, R6, R7==SP
22 Branching: Symbolic and Numeric supported
23
24 Notes:
25   Input files are assumed to end with a .asm extension
26   Immediate value sizes are checked
27   Instruction-less lines allowed
28   .ISR may be located anywhere in file
29   .define may be located anywhere, definition valid from location
   in file onwards, may replace existing definitions
30
31
32 Options:
33   -h, —help          show this help message and exit
34   -o FILE, —output=FILE
35                       output file for the assembled output

```

6.4 Error Messages

Code	Description
ERROR1	Instruction mnemonic 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 mnemonic 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 devices. 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.

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

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

```
1 uint16_t multi(uint16_t op1, op2){
2     uint16_t A,M,Q;
3     A = 0;
4     if(op1 < op2){                // Make M small, less loops
5         M = op1; Q = op2;
6     }else{
7         M = op2; Q = op1;
8     }
9     while(1){                    // No loop counter
10        if(M & 0x0001){           // LSB
11            A = A + Q;
12            if(A > 0xFFFF){        // Using carry flag
13                return 0;          // Overflow - fail
14            }
15        }
16        M = M >> 1;
17        if(0 == M){
18            return A;              // Finished - pass
19        }
20        if(Q & 0x8000){
21            return 0;              // Q >= 2^16 - fail
22        }
23    }
```

```

23     Q = Q << 1;
24 }
25 }

```

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

```

1  uint16_t fact(uint16_t x){
2      if(x == 0){
3          return 1;           // 0! = 1
4      }
5      return multi(x, fact(x-1)); // Recursive
6  }

```

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.

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.

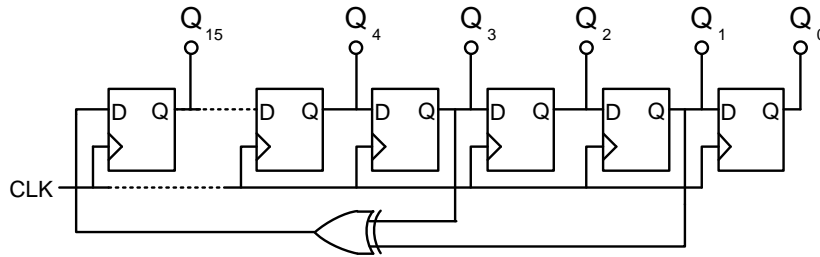


Figure 1: 16 Bit Linear Feedback Shift Register.

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

```

1 uint16_t rand(uint16_t last){
2     uint16_t next, test;
3     next = last >> 1;           // The shift
4     test = last << 2;           // Compare different bits
5     test = test ^ last;
6     if(test & 0x0008){          // Feedback to top
7         return (next | 0x8000);
8     }
9     return next;
10 }

```

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

```

1 #define TOP      0x0206
2 #define BOTTOM   0x0202
3 #define WRITE    0x0201
4 #define READ     0x0200

```

```

5 #define SERIAL 0xA000
6
7 isr () {
8     uint16_t data, readPtr, writePtr;
9     asm("DISI");           // critical op
10    data = read(SERIAL);
11    asm("ENAI");           // nested ints
12    readPtr = read(READ);
13    writePtr = read(WRITE);
14    if(((readPtr-1) == writePtr) ||
15        (readPtr == BOTTOM) ||
16        (writePtr == (TOP-1))) {
17        asm("RETI");        // full, don't write
18    }
19    if(readPtr == BOTTOM)
20        write(readPtr, data); // write to buffer
21    writePtr++;
22    if(writePtr == TOP) {
23        writePtr = BOTTOM;
24    } else {
25        writePtr++;
26    }
27    write(WRITE, writePtr);
28    asm("RETI");
29 }
30
31 void main() {
32     uint16_t readPtr, writePtr, data;
33     do {
34         readPtr = read(READ);
35         writePtr = read(WRITE);
36     } while(readPtr == writePtr)
37     data = read(readPtr)
38
39     fact();
40 }

```

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

```
1 // Hex file to specify serial data input
2 //
3 //
4 248    7
5 48F    6
6 6D6    5
7 91D    4
8 B64    7
9 DAB    5
10 36B1   3
11 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`
- `./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.

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

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 verified using the final design of the processor.

A.1 Multiply

Listing 7: multiply.asm

```
1      ADDIB R0,#0
2      ADDIB R0,#0
3      ADDIB R0,#0
4      ADDIB R0,#0
5      ADDIB R0,#0
6      ADDIB R0,#0
7      ADDIB R0,#0
8      ADDIB R0,#0
9      ADDIB R0,#0
10     ADDIB R0,#0
11     ADDIB R0,#0
12     ADDIB R0,#0
13     ADDIB R0,#0
14     ADDIB R0,#0
15     ADDIB R0,#0
16     ADDIB R0,#0
17     ADDIB R0,#0
18     ADDIB R0,#0
19     LUI    SP, #7          ; Init SP
20     LLI    SP, #208
21     LUI    R3, #8          ; SWs addr
22     LLI    R3, #0
23     LDW    R0,[R3,#0]      ; READ SWs
24     LUI    R1, #0
25     LLI    R1, #255        ; 0x00FF in R1
26     AND    R1,R0,R1        ; Lower byte SWs in R1
27     LSR    R0,R0,#8        ; Upper byte SWs in R0
28     PUSH   R0              ; Op1
29     PUSH   R1              ; Op2
30     SUB    R2,R2,R2        ; Zero required
31     PUSH   R2              ; Place holder is zero
32     BWL    .multi          ; Run Subroutine
33     POP    R1              ; Result
34     ADDIB  SP,#2           ; Dummy pop
35     ADDIB  R3,#1           ; Address of LEDS
```

```

36      STW      R1,[R3,#0]    ; Result on LEDS
37 .end      BR      .end      ; Finish loop
38 .multi    PUSH    R0
39          PUSH    R1
40          PUSH    R2
41          PUSH    R3
42          PUSH    R4
43          PUSH    R5
44          PUSH    R6
45          LDW     R0,[SP,#8]  ; R0 - Multiplier
46          LDW     R1,[SP,#9]  ; R1 - Quotient
47          CMP     R0,R1
48          BLT     .nSw        ; Branch if M < Q
49          ADDI    R2,R1,#0     ; Make M the smallest
50          ADDI    R1,R0,#0
51          ADDI    R0,R2,#0
52 .nSw      SUB     R2,R2,R2    ; R2 - Accumulator
53          ADDI    R3,R2,#1     ; R3 - 0x0001
54          LUI     R4,#128     ; R4 - 0x8000
55          LLI     R4,#0
56 .mloop    AND     R6,R0,R3    ; R6 - Cmp var
57          CMPI    R6,#1
58          BNE     .nAcc
59          SUB     R3,R3,R3
60          ADD     R2,R2,R1     ; A = A + Q
61          ADCI    R3,R3,#1
62          CMPI    R3,#2
63          BE      .fail       ; OV
64 .nAcc     LSR     R0,R0,#1    ; M = M >> 1
65          CMPI    R0,#0
66          BE      .done
67          AND     R5,R4,R1
68          CMPI    R5,#0
69          BNE     .fail
70          LSL     R1,R1,#1     ; Q = Q << 1
71          BR      .mloop
72 .done     STW     R2,[SP,#7]  ; Res on stack frame
73          POP     R6
74          POP     R5
75          POP     R4
76          POP     R3
77          POP     R2
78          POP     R1
79          POP     R0
80          RET

```

```

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

```

A.2 Factorial

Listing 8: factorial.asm

```

1      ADDIB R0,#0
2      ADDIB R0,#0
3      ADDIB R0,#0
4      ADDIB R0,#0
5      ADDIB R0,#0
6      ADDIB R0,#0
7      ADDIB R0,#0
8      ADDIB R0,#0
9      ADDIB R0,#0
10     ADDIB R0,#0
11     ADDIB R0,#0
12     ADDIB R0,#0
13     ADDIB R0,#0
14     ADDIB R0,#0
15     ADDIB R0,#0
16     ADDIB R0,#0
17     ADDIB R0,#0
18     ADDIB R0,#0
19     LUI   R7, #7
20     LLI   R7, #208
21     LUI   R0, #8      ; Address in R0
22     LLI   R0, #0
23     LDW   R1,[R0,#0]  ; Read switches into R1
24     PUSH  R1          ; Pass para
25     BWL   .fact      ; Run Subroutine
26     POP   R3          ; Para overwritten with result
27     ADDIB R0,#1
28     STW   R3,[R0,#0]  ; Result on LEDS
29 .end    BR     .end   ; finish loop
30 .fact  PUSH  R0
31        PUSH  R1
32        PUSH  LR
33        LDW   R1,[SP,#3] ; Get para
34        ADDIB R1,#0
35        BE    .retOne   ; 0! = 1
36        SUBI  R0,R1,#1
37        PUSH  R0        ; Pass para

```



```

38      BWL      .fact      ; The output remains on the stack
39      PUSH     R1         ; Pass para
40      SUBIB    SP,#1      ; Placeholder
41      BWL      .multi
42      POP      R1         ; Get res
43      ADDIB    SP,#2      ; pop x 2
44      STW      R1,[SP,#3]
45      POP      LR
46      POP      R1
47      POP      R0
48      RET
49 .retOne ADDIB    R1,#1    ; Avoid jump checking
50      STW      R1,[SP,#3]
51      POP      LR
52      POP      R1
53      POP      R0
54      RET
55 .multi  PUSH     R0
56      PUSH     R1
57      PUSH     R2
58      PUSH     R3
59      PUSH     R4
60      PUSH     R5
61      PUSH     R6
62      LDW      R0,[SP,#8] ; R0 - Multiplier
63      LDW      R1,[SP,#9] ; R1 - Quotient
64      CMP      R0,R1
65      BLT      .nSw      ; Branch if M < Q
66      ADDI     R2,R1,#0   ; Make M the smallest
67      ADDI     R1,R0,#0
68      ADDI     R0,R2,#0
69 .nSw    SUB     R2,R2,R2 ; R2 - Accumulator
70      ADDI     R3,R2,#1   ; R3 - 0x0001
71      LUI      R4,#128    ; R4 - 0x8000
72      LLI      R4,#0
73 .mloop  AND     R6,R0,R3 ; R6 - Cmp var
74      CMPI     R6,#1
75      BNE      .nAcc
76      SUB      R3,R3,R3
77      ADD      R2,R2,R1   ; A = A + Q
78      ADCI     R3,R3,#1
79      CMPI     R3,#2
80      BE       .fail     ; OV
81 .nAcc   LSR     R0,R0,#1 ; M = M >> 1
82      CMPI     R0,#0

```

```

83      BE      .done
84      AND     R5,R4,R1
85      CMPI    R5,#0
86      BNE     .fail
87      LSL     R1,R1,#1      ; Q = Q << 1
88      BR      .mloop
89 .done  STW     R2,[SP,#7]    ; Res on stack frame
90      POP     R6
91      POP     R5
92      POP     R4
93      POP     R3
94      POP     R2
95      POP     R1
96      POP     R0
97      RET
98 .fail  SUB     R2,R2,R2      ; OV - ret 0
99      BR      .done

```

A.3 Random

Listing 9: random.asm

```

1      ADDIB    R0,#0
2  ADDIB    R0,#0
3  ADDIB    R0,#0
4  ADDIB    R0,#0
5  ADDIB    R0,#0
6  ADDIB    R0,#0
7  ADDIB    R0,#0
8  ADDIB    R0,#0
9  ADDIB    R0,#0
10 ADDIB    R0,#0
11 ADDIB    R0,#0
12 ADDIB    R0,#0
13 ADDIB    R0,#0
14 ADDIB    R0,#0
15 ADDIB    R0,#0
16 ADDIB    R0,#0
17 ADDIB    R0,#0
18 ADDIB    R0,#0
19 ADDIB    R0,#0
20      LUI     SP,#7          ; Init SP
21      LLI     SP,#208
22      LUI     R0,#8          ; SW Address in R0

```

```

23      LLI      R0,#0
24      LDW      R1,[R0,#0] ; Read switches into R1
25      ADDIB    R0,#1      ; Address of LEDS in R0
26      PUSH     R1
27 .reset SUB     R4,R4,R4    ; Reset Loop counter
28 .loop  BWL     .rand
29      CMPI     R4,#15
30      BE       .write
31      ADDIB    R4,#1      ; INC loop counter
32      BR       .loop
33 .write LDW     R1,[SP,#0]  ; No pop as re-run
34      STW     R1,[R0,#0]  ; Result on LEDS
35      BR       .reset
36 .rand  PUSH     R0        ; LFSR Sim
37      PUSH     R1        ; Protect regs
38      PUSH     R2
39      LDW     R0,[SP,#3]  ; Last reg value
40      LSL     R1,R0,#2    ; Shift Bit 4 <- 2
41      XOR     R1,R0,R1    ; xor Gate
42      LSR     R0,R0,#1    ; Shifted reg
43      LUI     R2,#0
44      LLI     R2,#8
45      AND     R1,R2,R1    ; Mask off Bit 4
46      CMPI    R1,#0
47      BNE     .done
48      LUI     R1,#128
49      LLI     R1,#0
50      OR      R0,R0,R1    ; or with 0x8000
51 .done  STW     R0,[SP,#3]
52      POP     R2
53      POP     R1
54      POP     R0
55      RET

```

A.4 Interrupt

Listing 10: interrupt.asm

```

1      ADDIB    R0,#0
2      ADDIB    R0,#0
3      ADDIB    R0,#0
4      ADDIB    R0,#0
5      ADDIB    R0,#0
6      ADDIB    R0,#0

```

```

7      ADDIB R0,#0
8      ADDIB R0,#0
9      ADDIB R0,#0
10     ADDIB R0,#0
11     ADDIB R0,#0
12     ADDIB R0,#0
13     ADDIB R0,#0
14     ADDIB R0,#0
15     ADDIB R0,#0
16     ADDIB R0,#0
17     ADDIB R0,#0
18     ADDIB R0,#0
19     DISI                                ; Reset is off anyway
20     LUI      R7, #7
21     LLI      R7, #208
22     LUI      R0, #2                    ; R0 is read ptr      0x0200
23     LLI      R0, #0
24     ADDI     R1,R0,#2                  ; 0x0202
25     STW      R1,[R0,#0]                ; Read ptr set to 0x0202
26     STW      R1,[R0,#1]                ; Write ptr set to 0x0202
27     LUI      R0,#160                   ; Address of Serial control reg
28     LLI      R0,#1
29     LUI      R1,#0
30     LLI      R1,#1                    ; Data to enable ints
31     STW      R1,[R0,#0]                ; Store 0x001 @ 0xA001
32     ENAI
33     BR       .main
34 .isr     STF                                ; Keep flags, disable auto
35     PUSH     R0                        ; Save only this for now
36     LUI      R0,#160
37     LLI      R0,#0
38     LDW      R0,[R0,#0]                ; R1 contains read serial data
39     ENAI                                ; Don't miss event
40     PUSH     R1
41     PUSH     R2
42     PUSH     R3
43     PUSH     R4
44     LUI      R1,#2
45     LLI      R1,#0
46     LDW      R2,[R1,#0]                ; R2 contains read ptr
47     ADDI     R3,R1,#1
48     LDW      R4,[R3,#0]                ; R4 contain the write ptr
49     SUBIB    R2,#1                    ; Get out if W == R - 1
50     CMP      R4,R2
51     BE       .isrOut

```

```

52      ADDIB    R2,#1
53      LUI     R1,#2
54      LLI     R1,#2
55      CMP     R2,R1
56      BNE     .write
57      ADDIB    R1,#3
58      CMP     R4,R1
59      BE      .isrOut
60 .write  STW    R0,[R4,#0] ; Write to buffer
61      ADDIB    R4,#1
62      LUI     R1,#2
63      LLI     R1,#6
64      CMP     R1,R4
65      BNE     .wrapW
66      SUBIB    R4,#4
67 .wrapW  STW    R4,[R3,#0] ; Inc write ptr
68 .isrOut POP    R4
69      POP     R3
70      POP     R2
71      POP     R1
72      POP     R0
73      LDF
74      RETI
75 .main   LUI     R0, #2 ; Read ptr address in R0
76      LLI     R0, #0
77      LDW     R2,[R0,#0] ; Read ptr in R2
78      LDW     R3,[R0,#1] ; Write ptr in R3
79      CMP     R2,R3
80      BE      .main ; Jump back if the same
81      LDW     R3,[R2,#0] ; Load data out of buffer
82      ADDIB    R2,#1 ; Inc read ptr
83      SUB     R0,R0,R0
84      LUI     R0,#2
85      LLI     R0,#6
86      SUB     R0,R0,R2
87      BNE     .wrapR
88      SUBIB    R2,#4
89 .wrapR  LUI     R0, #2 ; Read ptr address in R0
90      LLI     R0, #0
91      STW     R2,[R0,#0] ; Store new read pointer
92      SUB     R4,R4,R4
93      LLI     R4,#15
94      AND     R3,R4,R3
95      CMPI    R3,#8
96      BE

```

```

97      LLI      R4,#7
98      AND      R3,R3,R4
99  .do    PUSH    R3
100     BWL      .fact
101     POP      R3
102     LUI      R4,#8
103     LLI      R4,#1      ; Address of LEDs
104     STW      R3,[R4,#0] ; Put factorial on LEDs
105     BR       .main      ; look again
106  .fact  PUSH    R0
107         PUSH    R1
108         PUSH    LR
109         LDW     R1,[SP,#3] ; Get para
110         ADDIB   R1,#0
111         BE      .retOne    ; 0! = 1
112         SUBI    R0,R1,#1
113         PUSH    R0          ; Pass para
114         BWL     .fact      ; The output remains on the stack
115         PUSH    R1          ; Pass para
116         SUBIB   SP,#1      ; Placeholder
117         BWL     .multi
118         POP     R1          ; Get res
119         ADDIB   SP,#2      ; pop x 2
120         STW     R1,[SP,#3]
121         POP     LR
122         POP     R1
123         POP     R0
124         RET
125  .retOne ADDIB   R1,#1      ; Avoid jump checking
126         STW     R1,[SP,#3]
127         POP     LR
128         POP     R1
129         POP     R0
130         RET
131  .multi  PUSH    R0
132         PUSH    R1
133         PUSH    R2
134         PUSH    R3
135         PUSH    R4
136         PUSH    R5
137         PUSH    R6
138         LDW     R0,[SP,#8] ; R0 - Multiplier
139         LDW     R1,[SP,#9] ; R1 - Quotient
140         CMP     R0,R1
141         BLT     .nSw      ; Branch if M < Q

```

```

142      ADDI    R2,R1,#0      ; Make M the smallest
143      ADDI    R1,R0,#0
144      ADDI    R0,R2,#0
145  .nSw      SUB    R2,R2,R2      ; R2 - Accumulator
146      ADDI    R3,R2,#1      ; R3 - 0x0001
147      LUI     R4,#128      ; R4 - 0x8000
148      LLI     R4,#0
149  .mloop    AND    R6,R0,R3      ; R6 - Cmp var
150      CMPI    R6,#1
151      BNE     .nAcc
152      SUB     R3,R3,R3
153      ADD     R2,R2,R1      ; A = A + Q
154      ADCI    R3,R3,#1
155      CMPI    R3,#2
156      BE      .fail      ; OV
157  .nAcc      LSR    R0,R0,#1      ; M = M >> 1
158      CMPI    R0,#0
159      BE      .done
160      AND     R5,R4,R1
161      CMPI    R5,#0
162      BNE     .fail
163      LSL     R1,R1,#1      ; Q = Q << 1
164      BR      .mloop
165  .done      STW     R2,[SP,#7]      ; Res on stack frame
166      POP     R6
167      POP     R5
168      POP     R4
169      POP     R3
170      POP     R2
171      POP     R1
172      POP     R0
173      RET
174  .fail      SUB     R2,R2,R2      ; OV - ret 0
175      BR      .done

```