# ELEC6027 - VLSI Design Project : Programmers Guide

Team R4

26th April, 2014

## Todo list

🟧 HSL - this doesn't sound right. Maybe "transfer of program flow". Not sure on the use of the word "control" but i know it is the technical term . . . 6
🟧 HSL - this is a bit too short. Surely there is more to say about it? . 6
🟧 HSL - I remember Iain saying something about the instruction formats being called A1 / A2. I don't see a problem personally as I can't remember exactly what he said! MRW - Said they should be diff instead of just A since there is no addressing mode field, numbered A1/A2 or redoing lettering is fine . . . 7
🟧 surely this we can just use the XOR instruction . . . 56
🟧 Maybe change to IEEE symbols if we have time . . . 57
🟧 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 . 60
🟧 how to run scan path sim too . . . 60
🟧 need to mention extracting? i wouldn't say so as it should already be done and that can't really change. . . . 60
🟧 and TCL? . . . 60
🟧 check that the dissembler works in all the simulations (it doesn't. . . ) 60

## Contents

**1 Introduction** **5**

**2 Architecture** **5**

**3 Register Description** **5**

**4 Instruction Set** **6**
4.1 General Instruction Formatting . . . 7
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** **61**

A.1 Multiply . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
A.2 Multiply Unrolled . . . . . . . . . . . . . . . . . . . . . . . . 62
A.3 Factorial . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
A.4 Random . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
A.5 Interrupt . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72

## 1 Introduction

Lorem Ipsum…

## 2 Architecture

Lorem Ipsum…

## 3 Register Description

Lorem Ipsum…

## 4 Instruction Set

The complete instruction set architecture includes a number of instructions for performing calculations on data, memory access, transfer of controlwithin a program and interrupt handling.

HSL - this doesn't sound right. Maybe "transfer of program flow". Not sure on the use of the word "control" but i know it is the technical term

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

HSL - this is a bit too short. Surely there is more to say about it?

## 4.1 General Instruction Formatting

HSL - I remember Iain saying something about the instruction formats being called A1 / A2. I don't see a problem personally as I can't remember exactly what he said! MRW - Said they should be diff instead of just A since there is no addressing mode field, numbered A1/A2 or redoing lettering is fine

| | 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 |
| 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 |
| immX: | Immediate value of length X |
| 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** |
|---|---|
| $\leftarrow$, $\rightarrow$ | Assignment |
| Result[$x$] | Bit $x$ of result |
| Ra[$x$ : $y$] | Bit range from $x$ to $y$ of register Ra |
| $+Ra$ | Positive value in Register Ra |
| $-Ra$ | Negative value in Register Ra |
| $<$ | Numerically greater than |
| $>$ | Numerically less 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 |
| (*cond*)? | Operation performed if *cond* evaluates to true |
| ! | 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 $\leftarrow$ | 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[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 ← | Ra + #imm5 |
| N ← | if (Result < 0) then 1, else 0 |
| Z ← | if (Result = 0) then 1, else 0 |
| V ← | if (Ra>0 and #imm5>0 and Result<0) or |
| | (Ra<0 and #imm5<0 and Result>0) then 1, else 0 |
| C ← | 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 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 ← | Rd + #imm8 |
| N ← | if (Result $< 0$) then 1, else 0 |
| Z ← | if (Result $= 0$) then 1, else 0 |
| V ← | if (Rd>0 and #imm8>0 and Result<0) or |
| | (Rd<0 and #imm8<0 and Result>0) then 1, else 0 |
| C ← | if (Result $> 2^{16} - 1$) or |
| | (Result $< -2^{16}$) 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 ← | Ra + Rb + C |
| N ← | if (Result $<$ 0) then 1, else 0 |
| Z ← | if (Result = 0) then 1, else 0 |
| V ← | 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 ← | 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**

| | |
|---|---|
| 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 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 $\leftarrow$ | 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[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**

$$\begin{array}{|r|l|}
\text{Rd} \leftarrow & \text{Rd - \#imm8} \\
\text{N} \leftarrow & \text{if (Result} < 0\text{) then 1, else 0} \\
\text{Z} \leftarrow & \text{if (Result} = 0\text{) then 1, else 0} \\
\text{V} \leftarrow & \text{if (Rd>0 and \#imm8>0 and Result<0) or} \\
 & \quad \text{(Rd<0 and \#imm8<0 and Result>0) then 1, else 0} \\
\text{C} \leftarrow & \text{if (Result} > 2^{16} - 1\text{) or} \\
 & \quad \text{(Result} < -2^{16}\text{) then 1, else 0}
\end{array}$$

**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 ← | Ra - #imm5 - C |
| N ← | if (Result $<$ 0) then 1, else 0 |
| Z ← | if (Result $=$ 0) then 1, else 0 |
| V ← | 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 ← | 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**

| | |
|---|---|
| | 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 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

| | |
|---|---|
| Rd ← | Ra `AND` Rb |
| N ← | if (Result $< 0$) then 1, else 0 |
| Z ← | if (Result $= 0$) then 1, else 0 |
| V ← | UNPREDICTABLE |
| C ← | 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 ← | Ra `OR` Rb |
| N ← | if (Result < 0) then 1, else 0 |
| Z ← | if (Result = 0) then 1, else 0 |
| V ← | UNPREDICTABLE |
| C ← | 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 ← | Ra `XOR` Rb |
| N ← | if (Result $<$ 0) then 1, else 0 |
| Z ← | if (Result $=$ 0) then 1, else 0 |
| V ← | UNPREDICTABLE |
| C ← | 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 ← | `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

| | |
|---|---|
| Rd ← | Ra `NAND` Rb |
| N ← | if (Result $<$ 0) then 1, else 0 |
| Z ← | if (Result $=$ 0) then 1, else 0 |
| V ← | UNPREDICTABLE |
| C ← | 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 ← | Ra `NOR` Rb |
| N ← | if (Result $<$ 0) then 1, else 0 |
| Z ← | if (Result $=$ 0) then 1, else 0 |
| V ← | UNPREDICTABLE |
| C ← | 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 ← | 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 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 ← | 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 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 ← | Mem[Ra + #imm5] |
| N ← | N |
| Z ← | Z |
| V ← | V |
| C ← | 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**

Mem[Ra + #imm5] $\leftarrow$ 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

$$\begin{array}{r|l|}
\text{Rd} \leftarrow & \{\#\text{imm8}, 0\} \\
\text{N} \leftarrow & \text{N} \\
\text{Z} \leftarrow & \text{Z} \\
\text{V} \leftarrow & \text{V} \\
\text{C} \leftarrow & \text{C}
\end{array}$$

### 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 ← | {Rd[15:8], #imm8} |
| N ← | N |
| Z ← | Z |
| V ← | V |
| C ← | 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 ← | PC + #imm8 |
| N ← | N |
| Z ← | Z |
| V ← | V |
| C ← | 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 ← PC + #imm8

| | |
|---|---|
| N ← | N |
| Z ← | Z |
| V ← | V |
| C ← | 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 ← PC + #imm8

| | |
|---|---|
| N ← | N |
| Z ← | Z |
| V ← | V |
| C ← | 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 ← | PC + 1; PC ← PC + #imm8 |
| N ← | N |
| Z ← | Z |
| V ← | V |
| C ← | 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 ← LR
 N ← N
 Z ← Z
 V ← V
 C ← 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 ← | Ra + #imm5 |
| N ← | N |
| Z ← | Z |
| V ← | V |
| C ← | 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 RL | eg. PUSH RL |

**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 RL | eg. POP RL |

**Operation**

| | |
|---|---|
| R7 ← | R7 + 1; Mem[R7] ← reg; |
| N ← | N |
| Z ← | Z |
| V ← | V |
| C ← | 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**

$$\begin{aligned} PC &\leftarrow \text{Mem}[R7] \\ N &\leftarrow N \\ Z &\leftarrow Z \\ V &\leftarrow V \\ C &\leftarrow C \end{aligned}$$

**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 hex. 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]
```

eg. `.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 can take either a symbolic or numeric value. Where a numeric must be relative and between -32 and 31 for a JMP instruction, or between -128 and 127 for any other branch type. If the branch exceeds the accepted range, the assembler will flag an error message.

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 and be no longer than 126 lines of code. Branches may occur within the ISR, but are not allowed into this subroutine 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 using the LUI and LLI instructions.

## 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 and then 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.

The subroutine operation is described using C in listing 2. If the result is greater than or equal to $2^{16}$ the subroutine will fail and return zero; The lowest bit of the multiplier control the accumulator and the overflow check. The multiplier is shifted right and the quotient is shifted left at every iteration. Equation (1) formally describes the result of algorithm disregarding physical limitations.

Listing 2: Multiply Subroutine

```c
uint16_t multi(uint16_t mul, quo){
    uint32_t A;
    uint16_t M,Q,i;
    A = 0; M = mul; Q = quo;
    for(i=0;i<16;i++){
        if(M && 0x0001){          // LSb
            A = A + Q;
            if(A > 0xFFFF){       // Larger than 16 bits?
                return 0;         // Fail
            }
        }
        Q = Q << 1;
        M = M >> 1;
    }
    return A;                     // Bottom 16 bits
}
```

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

The multiply code has also been optimised by unrolling the loop contents and is held in Appendix A.2 listing 8. In implementation a trade off between code size and execution time is made by loop unrolling the eight stages. This creates scope for optimisation in operations contained in the loop, doesn't use a counter and requires less branch operations.

## 7.2 Factorial

The code for the factorial program is held in Appendix A.3 listing 9. It is possible to calculate the factorial of any integer value between 0 and 8 inclusive. The main body of code masks the value read from the input switches so only acceptable values are passed to subroutine. The factorial subroutine is called which in turn calls the multiply subroutine discussed in section 7.1. The result is calculated recursively as described using C in listing 3.

Listing 3: Recursive Factorial Subroutine

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

## 7.3 Random

The code for the random program is held in Appendix A.4 listing 10. A random series of numbers is achieved by simulating a 16 bit linear feedback shift register. 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 is passed to the first subroutine call then using BWL instructions the parameter is altered and passed to the next subroutine call. No more PUSH or POP operations are performed. A load from the stack pointer is used write a new random number to LEDs. All contained within an unconditional branch.

An 2 input XOR gate is simulated by using masking the register value the comparing against inputs 00 and 11.These would return zero so only a shift is performed. If this is not true then a shift is performed followed by an OR operation with 0x8000 therefore feeding back a value to the top of the shift register. This is described using C in listing 4.

surely this we can just use the XOR instruction

Listing 4: Linear Feedback Shift Register Subroutine

```
uint16_t rand(uint16_t last){
    uint16_t next,test;
    next = last >> 1;                // Shift reg
```

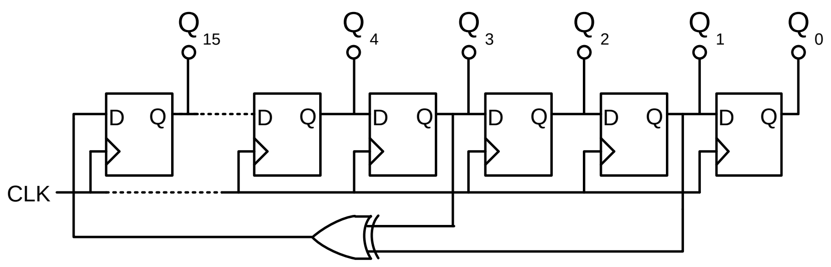

Figure 1: 16 Bit Linear Feedback Shift Register.

Maybe change to IEEE symbols if we have time

```
    test = last & 0x000A;                       // Bits 4 and 1
    if((test == 0x0000)|(test == 0x000A)){      // XOR test
        return next;
    }
    return next | 0x8000;                       // Feedback to top
}
```

## 7.4 Interrupt

The code for the interrupt program is held in Appendix A.5 listing 11. This is the most complex example and makes use of both the multiply and factorial subroutines in sections 7.1 and 7.2 respectively.

Listing 5: Serial Device Interrupt Service Request

```
isr(){

}

void main(){


}
```

# 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.

How to run for each of the behavioural, extracted and mixed

Before the simulator is invoked, the assembler should be invoked. 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

how to run scan path sim too

need to mention extracting? i wouldn't say so as it should already be done and that can't really change.

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.

and TCL?

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. 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.

check that the dissembler works in all the simulations (it doesn't. . . )

# 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

```
          LUI SP, #7        ; Init SP
          LLI SP, #208
          LUI R0, #8        ; SWs ADDR
          LLI R0, #0
          LDW R0,[R0,#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
          SUB R2,R2,R2      ; Zero required
          PUSH R0           ; Op1
          PUSH R1           ; Op2
          PUSH R2           ; Place holder is zero
          BWL .multi        ; Run Subroutine
          POP R1            ; Result
          ADDIB SP,#2       ; Duummy pop
          LUI R4, #8
          LLI R4, #1        ; Address of LEDS
          STW R1,[R4,#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
          SUB R2,R2,R2      ; R2 - Accumulator
          ADDI R3,R2,#1     ; R3 - Compare 1/0
          SUB R4,R4,R4      ; R4 - Loop counter
.lpMul    AND R6,R0,R3      ; R6 - Cmp var
          CMPI R6,#0
          BE .sh
```

```
         SUB R3,R3,R3
         ADD R2,R2,R1      ; A = A + Q
         ADCI R3,R3,#1
         CMPI R3,#2
         BE .over          ; OV
.sh      LUI R5,#128
         LLI R5,#0         ; 0x8000
         AND R5,R5,R1
         CMPI R5,#0
         BE .shift
         CMPI R0,#0
         BNE .over         ; And M != 0
.shift   LSL R1,R1,#1      ; Q = Q << 1
         LSR R0,R0,#1      ; M = M >> 1
         ADDIB R4,#1       ; i++
         CMPI R4,#15
         BNE .lpMul
.done    STW R2,[SP,#7]    ; Res on stack frame
         POP R6
         POP R5
         POP R4
         POP R3
         POP R2
         POP R1
         POP R0
         RET
.over    SUB R2,R2,R2      ; OV - RET 0
         BR .done
```

## A.2 Multiply Unrolled

Listing 8: multiplyUnroll.asm

```
         LUI SP, #7        ; Init SP
         LLI SP, #208
         LUI R0, #8        ; SWs ADDR
         LLI R0, #0
         LDW R0,[R0,#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
         SUB R2,R2,R2      ; Zero required
         PUSH R0           ; Op1
         PUSH R1           ; Op2
         PUSH R2           ; Place holder is zero
         BWL .multi        ; Run Subroutine
         POP R1            ; Result
         ADDIB SP,#2       ; Duummy pop
         LUI R4, #8
         LLI R4, #1        ; Address of LEDS
         STW R1,[R4,#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 R2,[SP,#8]    ; R2 - Multiplier
         LDW R3,[SP,#9]    ; R3 - Quotient
         SUB R4,R4,R4      ; R4 - Accumulator
         ADDI R6,R4,#1     ; R6 - Compare 1/0
         LUI R5,#128
         LLI R5,#0         ; R5 - 0x8000
         AND R1,R2,R6      ; Stage 1, R1 - cmp
         CMPI R1,#0        ; LSb ?
         BE .sh1
         ADD R4,R4,R3      ; (LSb == 1)?
.sh1     AND R0,R5,R3
         CMPI R0,#0
         BNE .over1
         LSL R3,R3,#1
         LSR R2,R2,#1
         AND R1,R2,R6      ; Stage 2
         CMPI R1,#0
         BE .sh2
         ADD R4,R4,R3
.sh2     AND R0,R5,R3
         CMPI R0,#0
         BNE .over1
         LSL R3,R3,#1
         LSR R2,R2,#1
         AND R1,R2,R6      ; Stage 3
         CMPI R1,#0
         BE .sh3
         ADD R4,R4,R3
.sh3     AND R0,R5,R3
```

```
        CMPI  R0,#0
        BNE .over1
        LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6      ; Stage 4
        CMPI  R1,#0
        BE .sh4
        ADD R4,R4,R3
.sh4    AND R0,R5,R3
        CMPI  R0,#0
        BNE .over1
        LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6      ; Stage 5
        CMPI  R1,#0
        BE .sh5
        ADD R4,R4,R3
.sh5    AND R0,R5,R3
        CMPI  R0,#0
        BNE .over1
        LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6      ; Stage 6
        CMPI  R1,#0
        BE .sh6
        ADD R4,R4,R3
.sh6    AND R0,R5,R3
        CMPI  R0,#0
        BNE .over1
        LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6      ; Stage 7
        CMPI  R1,#0
        BE .sh7
        ADD R4,R4,R3
.sh7    AND R0,R5,R3
        CMPI  R0,#0
        BNE .over1
        LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6      ; Stage 8
        CMPI  R1,#0
        BE .sh8
        ADD R4,R4,R3
        BR .sh8
```

```
.over1  BR .over
.sh8    AND R0,R5,R3
        CMPI R0,#0
        BNE .over
        LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6     ; Stage 9
        CMPI R1,#0
        BE .sh9
        SUB R6,R6,R6
        ADD R4,R4,R3
        ADCI R6,R4,#1
        CMPI R6,#2
        BNE .over
.sh9    AND R0,R5,R3
        CMPI R0,#0
        BNE .over
        LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6     ; Stage 10
        CMPI R1,#0
        BE .sh10
        SUB R6,R6,R6
        ADD R4,R4,R3
        ADCI R6,R4,#1
        CMPI R6,#2
        BNE .over
.sh10   AND R0,R5,R3
        CMPI R0,#0
        BNE .over
        LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6     ; Stage 11
        CMPI R1,#0
        BE .sh11
        SUB R6,R6,R6
        ADD R4,R4,R3
        ADCI R6,R4,#1
        BNE .over
.sh11   AND R0,R5,R3
        CMPI R0,#0
        BNE .over
        LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6     ; Stage 12
        CMPI R1,#0
        BE .sh12
        SUB R6,R6,R6
        ADD R4,R4,R3
        ADCI R6,R4,#1
        BNE .over
.sh12   AND R0,R5,R3
        CMPI R0,#0
        BNE .over
        LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6      ; Stage 13
        CMPI R1,#0
        BE .sh13
        SUB R6,R6,R6
        ADD R4,R4,R3
        ADCI R6,R4,#1
        BNE .over
.sh13   AND R0,R5,R3
        CMPI R0,#0
        BNE .over
        LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6      ; Stage 14
        CMPI R1,#0
        BE .sh14
        SUB R6,R6,R6
        ADD R4,R4,R3
        ADCI R6,R4,#1
        BNE .over
.sh14   AND R0,R5,R3
        CMPI R0,#0
        BNE .over
        LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6      ; Stage 15
        CMPI R1,#0
        BE .sh15
        SUB R6,R6,R6
        ADD R4,R4,R3
        ADCI R6,R4,#1
        BNE .over
.sh15   AND R0,R5,R3
        CMPI R0,#0
        BNE .over
        LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6     ; Stage 16
        CMPI R1,#0
        BE .sh16
        SUB R6,R6,R6
        ADD R4,R4,R3
        ADCI R6,R4,#1
        BNE .over
.sh16   STW R4,[SP,#7]   ; Res on stack frame
        POP R6
        POP R5
        POP R4
        POP R3
        POP R2
        POP R1
        POP R0
        RET
.over   SUB R4,R4,R4
        BR .sh16
```

## A.3 Factorial

Listing 9: factorial.asm

```
        LUI R7, #7
        LLI R7, #208
        LUI R0, #8       ; Address in R0
        LLI R0, #0
        LDW R0,[R0,#0]   ; Read switches into R0
        LUI R1,#0        ; Calculate only 8 or less
        LLI R1,#8
        CMP R1,R0
        BE .do
        SUBIB R1,#1
        AND R0,R0,R1
.do     PUSH R0          ; Pass para
        BWL .fact        ; Run Subroutine
        POP R0           ; Para overwritten with result
        LUI R4, #8
        LLI R4, #1       ; Address of LEDS
        STW R0,[R4,#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 R2,[SP,#8]   ; R2 - Multiplier
        LDW R3,[SP,#9]   ; R3 - Quotient
        SUB R4,R4,R4     ; R4 - Accumulator
        ADDI R6,R4,#1    ; R6 - Constant 1
        SUB R5,R5,R5     ; R5 - Constant 0
        SUB R0,R0,R0     ; R0 - C check
        AND R1,R2,R6     ; Stage 1, R1 - cmp
        CMPI R1,#0       ; LSb ?
        BE .sh1
        ADD R4,R4,R3     ; (LSb == 1)?
.sh1    LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6     ; Stage 2
        CMPI R1,#0
```

```
          BE .sh2
          ADD R4,R4,R3
.sh2      LSL R3,R3,#1
          LSR R2,R2,#1
          AND R1,R2,R6     ; Stage 3
          CMPI R1,#0
          BE .sh3
          ADD R4,R4,R3
.sh3      LSL R3,R3,#1
          LSR R2,R2,#1
          AND R1,R2,R6     ; Stage 4
          CMPI R1,#0
          BE .sh4
          ADD R4,R4,R3
.sh4      LSL R3,R3,#1
          LSR R2,R2,#1
          AND R1,R2,R6     ; Stage 5
          CMPI R1,#0
          BE .sh5
          ADD R4,R4,R3
.sh5      LSL R3,R3,#1
          LSR R2,R2,#1
          AND R1,R2,R6     ; Stage 6
          CMPI R1,#0
          BE .sh6
          ADD R4,R4,R3
.sh6      LSL R3,R3,#1
          LSR R2,R2,#1
          AND R1,R2,R6     ; Stage 7
          CMPI R1,#0
          BE .sh7
          ADD R4,R4,R3
.sh7      LSL R3,R3,#1
          LSR R2,R2,#1
          AND R1,R2,R6     ; Stage 8
          CMPI R1,#0
          BE .sh8
          ADD R4,R4,R3
.sh8      LSL R3,R3,#1
          LSR R2,R2,#1
          AND R1,R2,R6     ; Stage 9
          CMPI R1,#0
          BE .sh9
          ADD R4,R4,R3
          ADCI R0,R5,#0
```

```
        CMPI R0,#0
        BNE .over
.sh9    LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6      ; Stage 10
        CMPI R1,#0
        BE .sh10
        ADD R4,R4,R3
        ADCI R0,R5,#0
        CMPI R0,#0
        BNE .over
.sh10   LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6      ; Stage 11
        CMPI R1,#0
        BE .sh11
        ADD R4,R4,R3
        ADCI R0,R5,#0
        CMPI R0,#0
        BNE .over
.sh11   LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6      ; Stage 12
        CMPI R1,#0
        BE .sh12
        ADD R4,R4,R3
        ADCI R0,R5,#0
        CMPI R0,#0
        BNE .over
.sh12   LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6      ; Stage 13
        CMPI R1,#0
        BE .sh13
        ADD R4,R4,R3
        ADCI R0,R5,#0
        CMPI R0,#0
        BNE .over
.sh13   LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6      ; Stage 14
        CMPI R1,#0
        BE .sh14
        ADD R4,R4,R3
        ADCI R0,R5,#0
        CMPI  R0,#0
        BNE .over
.sh14   LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6     ; Stage 15
        CMPI  R1,#0
        BE .sh15
        ADD R4,R4,R3
        ADCI  R0,R5,#0
        CMPI  R0,#0
        BNE .over
.sh15   LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6     ; Stage 16
        CMPI  R1,#0
        BE .sh16
        ADD R4,R4,R3
        ADCI  R0,R5,#0
        CMPI  R0,#0
        BNE .over
.sh16   STW R4,[SP,#7]   ; Res on stack frame
        POP R6
        POP R5
        POP R4
        POP R3
        POP R2
        POP R1
        POP R0
        RET
.over   SUB R4,R4,R4
        STW R4,[SP,#7]   ; Res on stack frame
        POP R6
        POP R5
        POP R4
        POP R3
        POP R2
        POP R1
        POP R0
        RET
```

## A.4 Random

Listing 10: random.asm

```
         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.5 Interrupt

Listing 11: interrupt.asm

```
         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    DISI
        STF               ; Keep flags
        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 R2,[SP,#8]  ; R2 - Multiplier
         LDW R3,[SP,#9]  ; R3 - Quotient
         SUB R4,R4,R4    ; R4 - Accumulator
         ADDI R6,R4,#1   ; R6 - Constant 1
         SUB R5,R5,R5    ; R5 - Constant 0
         SUB R0,R0,R0    ; R0 - C check
         AND R1,R2,R6    ; Stage 1, R1 - cmp
         CMPI R1,#0      ; LSb ?
         BE .sh1
         ADD R4,R4,R3    ; (LSb == 1)?
.sh1     LSL R3,R3,#1
         LSR R2,R2,#1
         AND R1,R2,R6    ; Stage 2
         CMPI R1,#0
         BE .sh2
         ADD R4,R4,R3
.sh2     LSL R3,R3,#1
         LSR R2,R2,#1
```

```
        AND R1,R2,R6       ; Stage 3
        CMPI R1,#0
        BE .sh3
        ADD R4,R4,R3
.sh3    LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6       ; Stage 4
        CMPI R1,#0
        BE .sh4
        ADD R4,R4,R3
.sh4    LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6       ; Stage 5
        CMPI R1,#0
        BE .sh5
        ADD R4,R4,R3
.sh5    LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6       ; Stage 6
        CMPI R1,#0
        BE .sh6
        ADD R4,R4,R3
.sh6    LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6       ; Stage 7
        CMPI R1,#0
        BE .sh7
        ADD R4,R4,R3
.sh7    LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6       ; Stage 8
        CMPI R1,#0
        BE .sh8
        ADD R4,R4,R3
.sh8    LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6       ; Stage 9
        CMPI R1,#0
        BE .sh9
        ADD R4,R4,R3
        ADCI R0,R5,#0
        CMPI R0,#0
        BNE .over
.sh9    LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6      ; Stage 10
        CMPI R1,#0
        BE .sh10
        ADD R4,R4,R3
        ADCI R0,R5,#0
        CMPI R0,#0
        BNE .over
.sh10   LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6      ; Stage 11
        CMPI R1,#0
        BE .sh11
        ADD R4,R4,R3
        ADCI R0,R5,#0
        CMPI R0,#0
        BNE .over
.sh11   LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6      ; Stage 12
        CMPI R1,#0
        BE .sh12
        ADD R4,R4,R3
        ADCI R0,R5,#0
        CMPI R0,#0
        BNE .over
.sh12   LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6      ; Stage 13
        CMPI R1,#0
        BE .sh13
        ADD R4,R4,R3
        ADCI R0,R5,#0
        CMPI R0,#0
        BNE .over
.sh13   LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6      ; Stage 14
        CMPI R1,#0
        BE .sh14
        ADD R4,R4,R3
        ADCI R0,R5,#0
        CMPI R0,#0
        BNE .over
.sh14   LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6     ; Stage 15
        CMPI R1,#0
        BE .sh15
        ADD R4,R4,R3
        ADCI R0,R5,#0
        CMPI R0,#0
        BNE .over
.sh15   LSL R3,R3,#1
        LSR R2,R2,#1
        AND R1,R2,R6     ; Stage 16
        CMPI R1,#0
        BE .sh16
        ADD R4,R4,R3
        ADCI R0,R5,#0
        CMPI R0,#0
        BNE .over
.sh16   STW R4,[SP,#7]   ; Res on stack frame
        POP R6
        POP R5
        POP R4
        POP R3
        POP R2
        POP R1
        POP R0
        RET
.over   SUB R4,R4,R4
        STW R4,[SP,#7]   ; Res on stack frame
        POP R6
        POP R5
        POP R4
        POP R3
        POP R2
        POP R1
        POP R0
        RET
```