Final Report
ELEC6027: VLSI Design Project

Team R4
Henry Lovett (hl13g10)
Ashley Robinson (ajr2g10)
Martin Wearn (mw20g10)

13th May, 2014

Contents

1. Introduction

This report documents the design and test of the Samurai processor. It was designed for the ELEC6027: VLSI design module by Team R4.

The report is broken down into four main sections. Firstly, the overall design of the architecture and datapath hardware is discussed. This is followed by the instruction set considerations and machine code formatting choices. The design of the processor, including all datapath modules and the behavioural model for the controller, is then discussed. This includes state machines and circuit diagrams where applicable. Following the design, testing strategies are explained and discussed. This includes descriptions of the tests conducted on all designed modules. Finally, the report evaluates the success of the final design with some concluding remarks.

A comprehensive list of instructions and project management considerations are also included in the appendix. Along with a division of labour form.

1.1 Architecture

The architecture for the processor was initially base upon a MIPS style datapath. Support was then added to allow the ARM Thumb instruction set (see Chapter 2) to be executed on the datapath. Extra aspects of the datapath were then added based on the research done. The full datapath diagram is illustrated in Figure 1.1.

A dedicated Link Register is used to improve the speed when calling leaf functions. The original design also included a dedicated Stack Pointer, however this was later removed during the project due to the need for specialist instructions. The convention of using Register 7 as the Stack Pointer was instead used. Both of these aspects were added from the recommendations from the research report on subroutines.

The processor supports four status flags: carry, overflow, negative and zero. Whilst being based upon a load/store architecture with up to 3 operands. There are eight general purpose registers without a zero register, and separate link register and program counter.

The datapath was also modified during the project to allow for interrupt support. These changes included an input to the Program Counter to jump to a specific location, reading and writing of the status register from and to the system bus, and input of the Program Counter directly from the system bus.

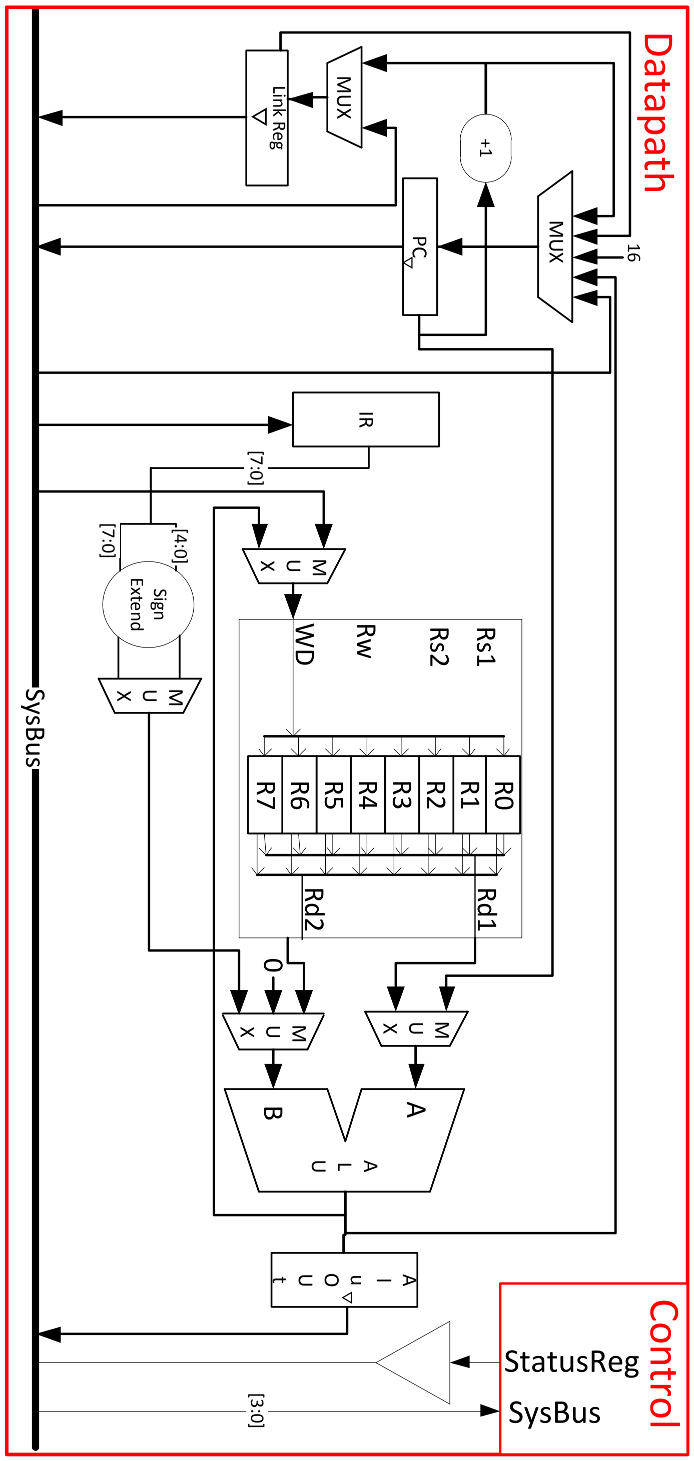


Figure 1.1: The Architecture diagram of the processor.

2. Instruction Set

In designing the instruction set architecture (ISA), emphasis was put on creating a complete set of basic operations which could be used to implement any program. Early research suggested a RISC based architecture would be suitable since they have a small number of instructions and are optimised for a smaller chip area. They also promote a simpler datapath since the same length instructions can result in easier bit slicing. Irregular lengths would cause common fields to be in a different location within the instruction, leading to more complex decoding and potential wasted hardware when executing shorter instructions.

Research into different microprocessor instruction sets has highlighted the potential of basing this system on the ARM Thumb architecture. This is a 16 bit subset of the main 32 bit ARM instruction set which contains a complete set of instructions. The full Thumb ISA was not used to aid simplicity and because it supports operations not available within the intended datapath. A summary of the final instruction set is provided in Appendix A. The number of operations taken from ARM was adapted for greater support and novelty instructions. This included: use of carry flag with immediate values, a completed logic set, loading registers, better branching and interrupt support. **LUI** and **LLI** provide the ability to set the upper or lower byte of a register to an immediate value for initialisation. It was also decided to simplify subroutine returns by incorporating a **RET** instruction as supported by the SPARC microprocessor, and allow transferring of control to an explicit location in memory using the **JMP** instruction. Instructions were added for enabling or disabling interrupts, loading/storing status flags and an interrupt specific subroutine return.

Within this ISA it was decided to support up to three operands. This allows greater flexibility with the instructions available and reduces the amount of memory required to perform data processing operations. The size of each field was determined by the number of possibilities needed and a length of 16 bits per instruction. The number of instructions and their groupings determined the requirement of having 6 bits for the Opcode field. As such, 8 internal registers could be used since it is a realistic number for a RISC system and can be addressed in the remaining 10 bits. With the option of expanding the third operand to a 5 bit immediate value, benefiting from the higher frequency of smaller immediate values over larger ones. There was also support added for byte sized operations with two operand formatting since this is a standard length for small binary values.

An important aspect of ISA design is the consideration of how much memory is required to store a particular program. An architecture which requires less space will be desirable since more information can be stored in the same amount of memory. To achieve this a high code density is required, as shown during instruction set research. The density of this system has been improved by using three operand instructions to reduce the number of data transfers required. This is illustrated in Figure 2.1 in terms of register transfers required to add two values and place the result in another register. Further improvements have been made by supporting multi-bit shifting, this means the useful maximum of a 15 bit shift can be done in one operation.

3 Operands	R1 ← R2 + R3
2 Operands	R1 ← R2
	R1 ← R1 + R3
1 Operand	Acc ← R2
	Acc ← Acc + R3
	R1 ← Acc

Figure 2.1: Comparison of Operand Amounts

Both control transfer and interrupt operations, with one exception, do not use any registers. This meant a single Opcode could be used to define an instruction type, with an additional field for defining the explicit instruction to perform. A 3 bit condition field allows a sufficient quantity of branching operations to be supported, leaving 8 bits to define the distance to move forwards or backwards. While an additional 3 bit field in interrupt operations maintains location consistency with control transfer and is long enough to define the 5 instructions needed.

To promote orthogonality, the instruction formatting for data manipulation operations followed a similar structure to the ARM Thumb, as shown in Table 2.1. This was adapted for all other instruction types, and reordered to ensure immediate values were always on the far right of the instruction. This was to make sign extension of immediate values in the datapath easier since they are always in the same location. It was also necessary to align all the destination and source registers to maintain consistency between instructions, aiding datapath simplicity.

Allocation of Opcodes was done using K-map design with the arrangement designated according to the operation needing to be performed within the ALU module. This was because this allocation would have the greatest effect on the amount of logic needed for the ALU decoder. The resultant mapping is shown in Table 2.2, with the important groupings highlighted. The four groups shown correspond to some command signals between the ALU and decoder which need to be active for more than one instruction.

Table 2.1: 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
A2	**Data Manipulation**	**Immediate**	Opcode					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	**Control Transfer**	**Jump**	1	1	1	1	0	Cond.			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

Allocation of condition codes for control transfer instructions were based upon the type and action of each branch. This helps to simplify decoding in the controller, promoting a smaller synthesized layout. The aspects considered were: conditional or unconditional, use of link register and flags used. These are summarised in Table 2.3a. From this, the first bit was set according to whether there was a condition to be checked. Then the second bit was set if the unconditional instruction used the link register, or the conditional instruction checked the zero flag. Since interrupts are only used in the controller, and added as they were deemed necessary, there is no specific ordering to the operations. The code assignments are shown in Table 2.3b.

Table 2.2: Opcode Assignment K-Maps

	00	01	11	10
000	LDW	STW	NOP	AND
001	POP	PUSH	‘F’	OR
011	ADDIB	SUBIB	▨	XOR
010	ADD	SUB	NEG	NOT
110	ADDI	SUBI	‘D’	NAND
111	CMP	CMPI	LSL	NOR
101	ADCI	SUCI	LSR	LLI
100	ADC	SUC	ASR	LUI

● FAOut ● ShOut

	00	01	11	10
000	LDW	STW	NOP	AND
001	POP	PUSH	‘F’	OR
011	ADDIB	SUBIB	▨	XOR
010	ADD	SUB	NEG	NOT
110	ADDI	SUBI	‘D’	NAND
111	CMP	CMPI	LSL	NOR
101	ADCI	SUCI	LSR	LLI
100	ADC	SUC	ASR	LUI

● SUB ● ShR

Table 2.3: Condition Code Assignments

(a) Cond. Assignment

	Un	LR	Z	N,V	Cond.
BR	✓	✗	✗	✗	000
BNE	✗	✗	✓	✗	110
BE	✗	✗	✓	✗	111
BLT	✗	✗	✗	✓	100
BGE	✗	✗	✗	✓	101
BWL	✓	✓	✗	✗	011
RET	✓	✓	✗	✗	010
JMP	✓	✗	✗	✗	001

(b) ICond. Assignment

	ICond.
RETI	000
ENAI	001
DISI	010
STF	011
LDF	100

3. Design and Implementation

3.1 Register Block

The register block contains eight general purpose registers. No dummy register is implemented in the register block. This module has three address inputs, two read addresses and one write address. There are two read outputs, which output the stored data, and a data input to be written. Reads are asynchronous and must always be driven. Writes are synchronous and must implement a write enable signal.

To optimise the design for space, the module is broken down into a bit slice and a decoder. The whole block is made up of sixteen slices and one decoder.

3.1.1 Bit Slice Design

The bit slice consists of eight load enable registers. Two tristate buffers are used per register onto different buses for the two read data outputs. The registers are selected by one hot encoding and as such will result in the output bus always being driven. The write enables are also controlled using one hot encoding. However, all signals are allowed to be off if no register is to be written to. The circuit diagram for the bit slice is shown in Figure 3.1.

The read decoder is based around a 3-8 decoder. For each 3 bit input, only one output should be high. The logic functions used are summarised in Table 3.1. The bit sliced circuit diagram is shown in Figure 3.2. This illustrates only one of the decoders needed since the remainder are identical. However, the write decoder

Table 3.1: Decoder Logic Functions

Register Number	Logic Select Function
0	$!(Rs0 \vert Rs1 \vert Rs2)$
1	$!(Rs1 \vert Rs2)\&Rs0$
2	$!(Rs2 \vert Rs0)\&Rs1$
3	$!(!(Rs0\&Rs1) \vert Rs2)$
4	$!(Rs0 \vert Rs1)\&Rs2$
5	$!(!(Rs0\&Rs2) \vert Rs1)$
6	$!(!(Rs1\&Rs2) \vert Rs0)$
7	$Rs0\&Rs1\&Rs2$

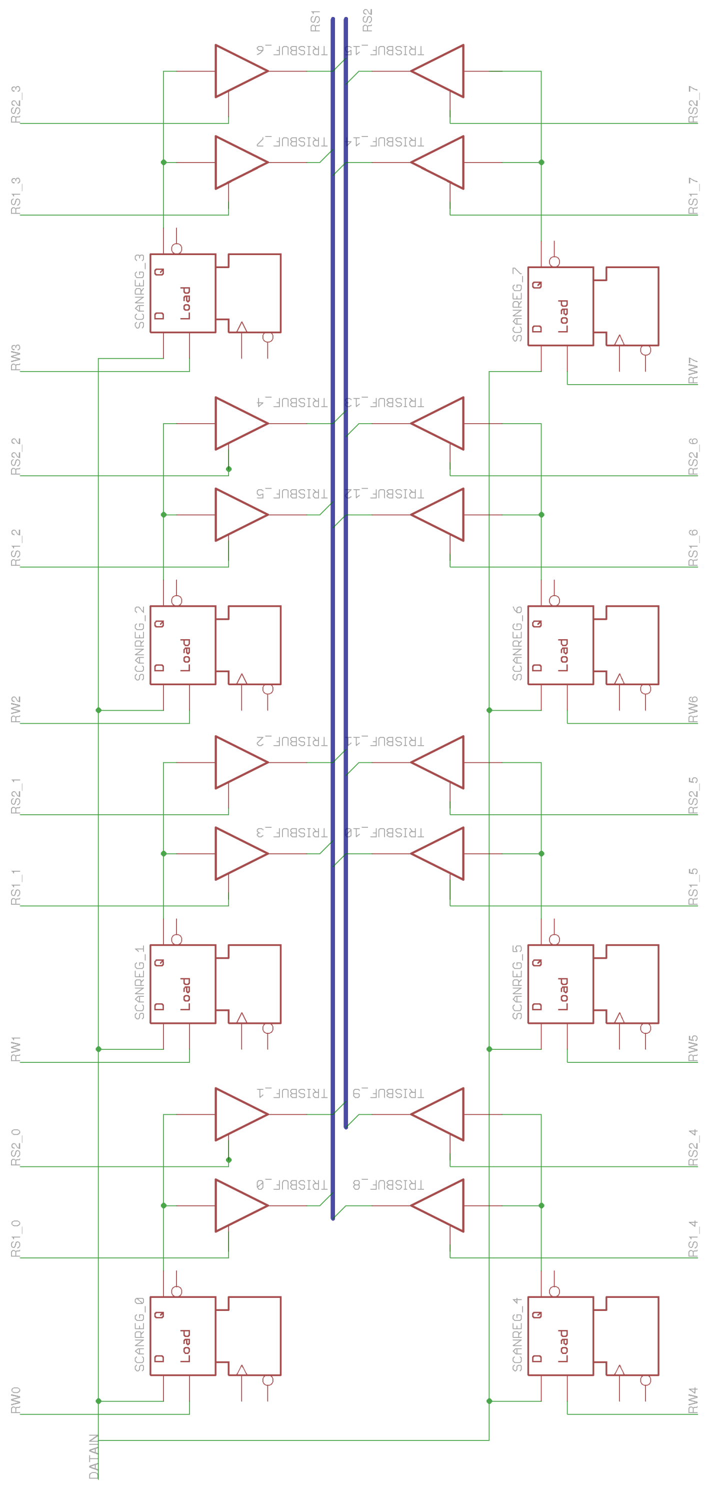


Figure 3.1: Circuit diagram for the register block slice.

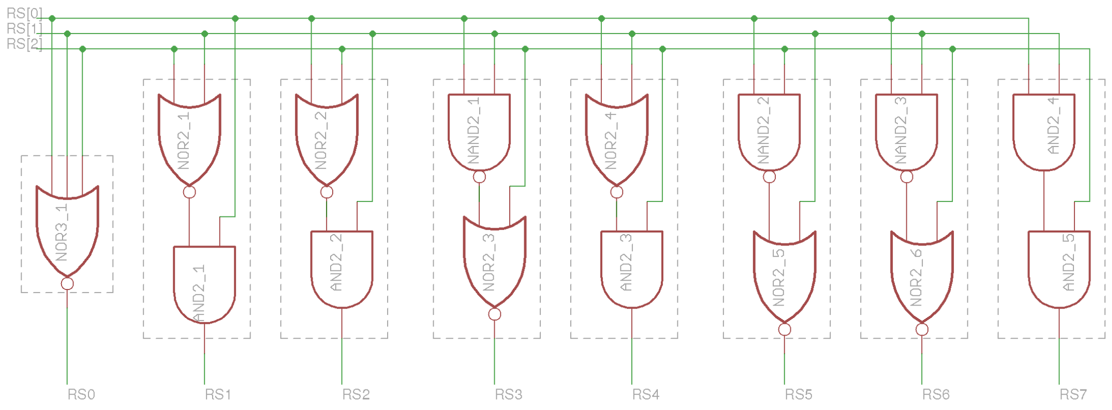


Figure 3.2: Circuit diagram for the register decoder.

must also use a write enable signal. This is done by using the same logic functions shown in Table 3.1, but the resulting signals are then ANDed with the register write enable signal.

The layout in silicon consisted of an array of slices. All I/Os for the slice are then in the same position and all global signals from the decoder are propagated through the slice. The use of tristate buffers reduces the wiring needed in the design. The decoder was designed to fit above the block of general purpose registers. All three decoders were distributed in the implementation and spaced to reduce the size of the wiring channel.

3.2 Program Counter and Link Register

The program counter and link register were implemented in one module. The module has two load enable registers. The data source logic and an adder for incrementing the program counter are also implemented in this module.

The program counter sources are:

1. Incremented program counter - for normal sequential operation
2. Link Register - for returning from a subroutine call
3. ALU output - for a jump to a register contents
4. System bus - for returning the program counter from the stack after an interrupt service routine.

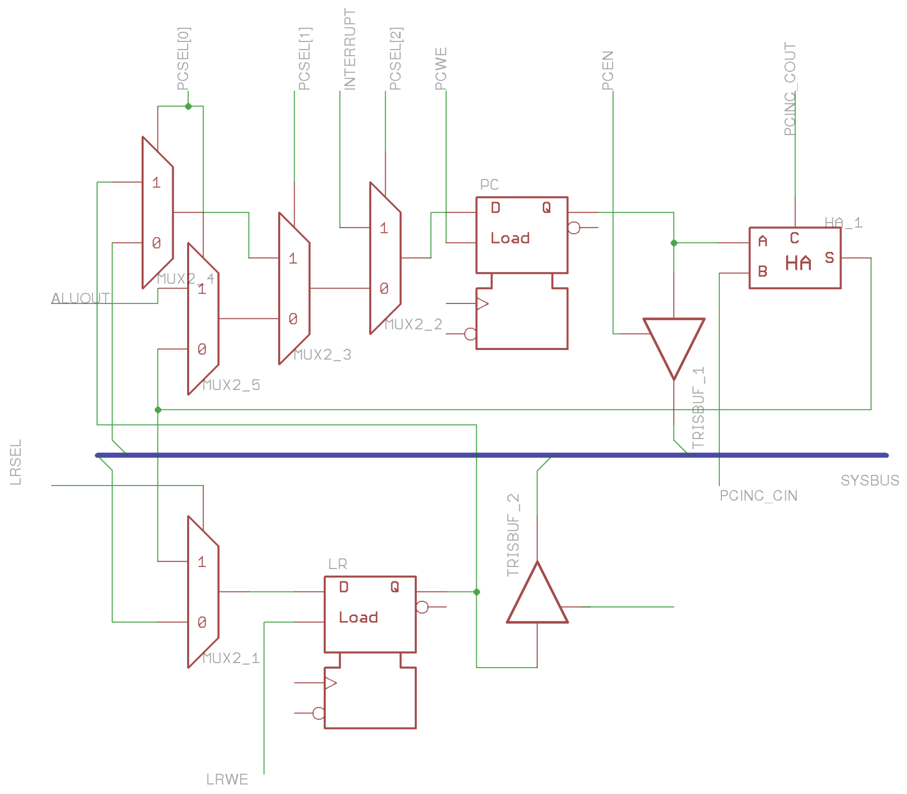


Figure 3.3: Circuit Diagram for Program Counter and Link Register block slice.

5. ISR Location - a constant defined for the location of the interrupt service routine in memory.

The link register has two inputs:

1. Incremented program counter - for storing the return address when executing a function call

2. System Bus - for retrieving a return address from the stack

This module is made up of sixteen identical bitslices. All control signals are connected to the controller. The circuit diagram is shown in Figure 3.3. The slice is implemented to match the circuit diagram. Multiplexors are used instead of tristate buffers since there are few data sources used which are not significantly distributed around the datapath to require a bus. It also removes the need for a decoder.

The module is made up of an array of slices. The control signals are propagated throughout the module vertically. The system bus connection has also been added and the carry out and the second input to the half adder must be connected between the slices.

3.3 Instruction Register

The instruction register module has a load enable register for storing of the instruction. It also has the immediate selection and sign extension circuitry. The two immediate values are either a five bit signed value located at position [4:0] of the instruction, or an eight bit signed value at [7:0] of the instruction. Only the **LLI** instruction does not use a signed immediate value. This is implemented by the ALU, discussed in Section 3.4.

The instruction register module is separated into three different bit slices. The circuit diagrams for these are shown in Figure 3.4. It cannot be broken into sixteen identical modules due to data in the instruction needing sign extension.

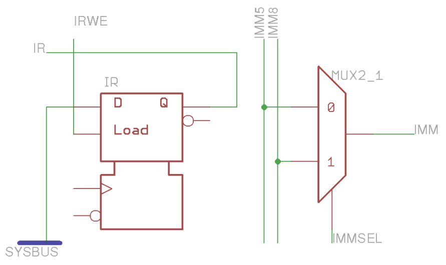
The data for the instruction register is only read from the system bus. The instruction is then directly fed out of this module to the controller or register decoding blocks. The three blocks differ due to the sign extension and are referred to as “AA”, “BA” or “BB”. The three modules contain the same cells, but the input and output wires to the multiplexor differ.

AA - The first block chooses the sign from either the eight or five bit immediate. The sign is then passed in at the bottom of the module and propagated to the top.

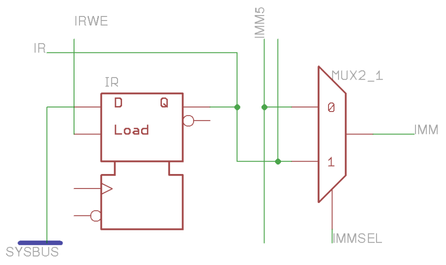
BA - This block chooses between the instruction register value or a value passed into the bottom of the module. This input is the sign of the five bit immediate and is outputted to the top of the module.

BB - This block takes the value of the instruction register and passes it to both of the multiplexor inputs. The multiplexor is redundant here but kept to keep the size of the three modules identical. All inputs to the bottom of this module are ignored.

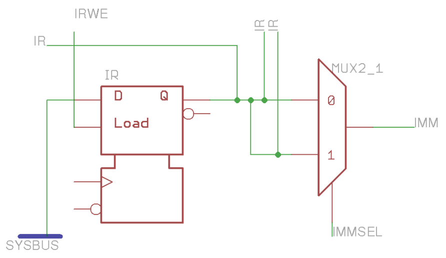
By stacking these modules using 5×BB, 3×BA and 8×AA (bottom to top), the sign bits will be propagated at the correct points. The output is then a full sixteen bit signed value of the relevant immediate selected by the *ImmSel* signal.



(a) The AA type slice



(b) The BA type slice



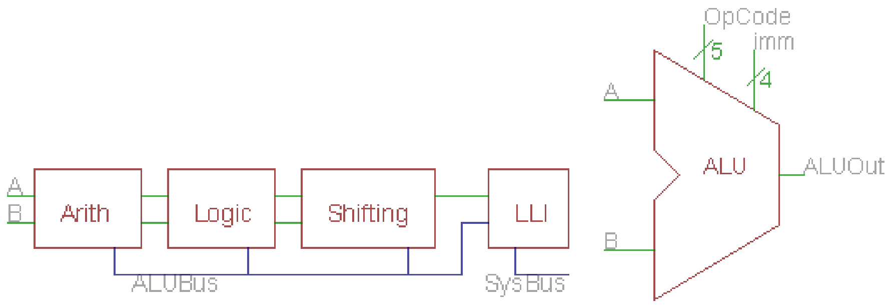
(c) The BB type slice

Figure 3.4: The three circuits used for the instruction register block.

3.4 Arithmetic Logic Unit

The ALU is the central unit for performing calculations within the datapath. Most instructions use the ALU as part of their operation and as such this module needs to interpret every instruction and perform the necessary function. The functions implemented fall into one of four types: arithmetic, logic, shifting and load lower. Arithmetic operations are centralised around a full adder. Additional gates are used for subtraction and flag calculations. The logic operations consists of the gate corresponding to each of the logic instructions supported. Shifts are performed using a logarithmic shifter to support up to a 15 bit shift in one clock cycle. Finally, the LLI module concatenates the upper byte of the destination register with the provided 8 bit immediate value. The breakdown of the ALU is illustrated in Figure 3.5a, with the output of each section connected to an internal bus through a number of tristate buffers.

This design then needed to be bit sliced to simplify the datapath and promote hierarchical construction. It was decided to add an additional input to the module for the 4 bit immediate value required to define the shifting amount. Otherwise the lowest four slices of the ALU would each need different routing to each segment,



(a) Modular ALU Layout (b) Top-Level View of ALU

Figure 3.5: ALU Abstraction

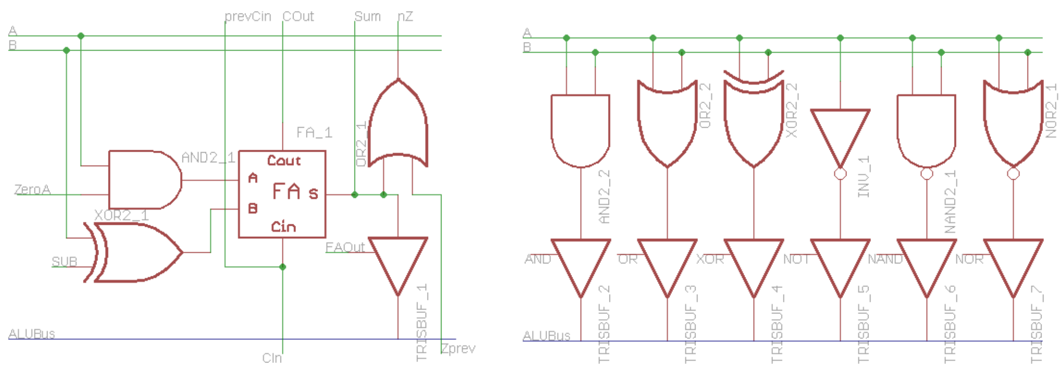
reducing how much of each slice could be identically duplicated.

Initial design of the ALU module was based upon the planned datapath layout as shown in Figure 3.5b. From this it was seen that there may be some difficulty with interpreting the Opcodes provided. Each slice would have to individually interpret the Opcode, using much more space than is necessary due to replicated hardware. Observations showed that different groups of instructions would perform the same operation within the ALU, therefore a separate decoder module was implemented to reduce required area.

3.4.1 ALU Slice

Due to the regularity of the ALU, the bit slice was simple to implement. The zero flag was calculated using an OR gate in each slice, which is then inverted in the decoder. Carry and negative flags are available using the outputs of the top slice. The overflow flag is calculated using the top slice outputs in the decoder. The arithmetic implementation is shown in Figure 3.6a. The logic operations were simple to implement, using the logic gates, each with a tristate buffer to the output bus. The circuit diagram is shown in Figure 3.6b.

Implementing shifting capabilities into individual bits required few logic gates since it is a wire-dominated circuit, but each wire needed to be lined up correctly between slices. This results in this slice section having more dependency on the neighbouring slices than both arithmetic and logic functions. The left and right shifting have been implemented using separate hardware, with a logarithmic strategy. To support arithmetic shifting, the shifted value into a right shift can be either a zero or the current operand's sign. Implementation of this section is



(a) Arithmetic Section (b) Logic Section

Figure 3.6: Bitsliced Circuit Diagrams for Arithmetic and Logic Sections

shown in Figure 3.7.

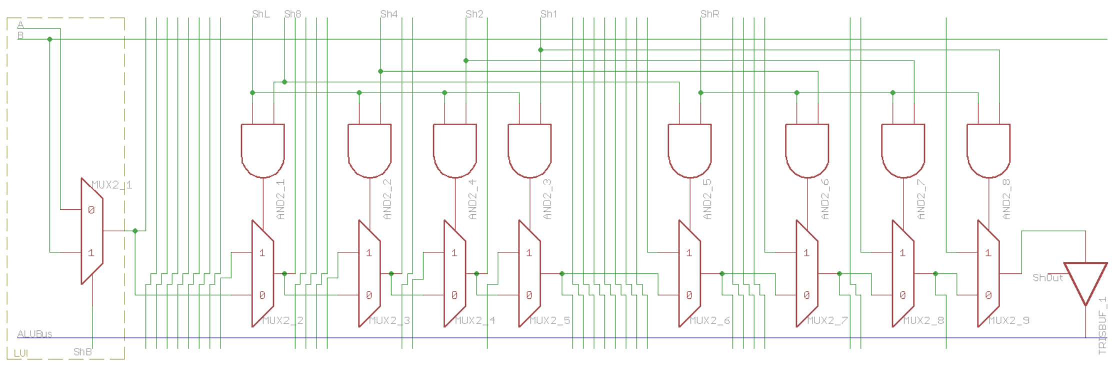


Figure 3.7: Bitsliced Circuit Diagram for Shift Section of ALU

Every instruction can be carried out using one of the previous sections, with the exception of **LUI** and **LLI**. Loading an upper immediate value involves concatenating the 8 bit value with 8 zero bits. This is equivalent to shifting the second operand by 8 bits, as such it can be implemented with an additional multiplexor before the shifting section to select between each input operand. This is shown in Figure 3.7 as a precursor to standard shifting operation. Loading a lower immediate involves concatenating the existing high byte of the destination register with the value in the instruction. However, it is not possible to separate this into 16 identical slices. As such two versions of the slice were designed, one which passes through the upper byte with no change, and one which selects between the lower byte of the input register value and the immediate value. These form modules which are separate to the main slice consisting of either wiring or a multiplexor.

3.4.2 ALU Decoder

The purpose of a decoder module is to convert any given Opcode into a number of signals to control the path of data within the ALU. As such it is composed of combinational logic. Table 3.2 shows the necessary control signals for each mnemonic.

Table 3.2: Control outputs for each available instruction mnemonic

Instruction		Decoder Outputs
LDW	00000	FAOut
POP	00001	FAOut
ADDIB	00011	FAOut
ADD	00010	FAOut
ADDI	00110	FAOut
CMP	00111	FAOut, SUB
ADCI	00101	FAOut, *UseC*
ADC	00100	FAOut, *UseC*
STW	01000	FAOut
PUSH	01001	FAOut, SUB
SUBIB	01011	FAOut, SUB
SUB	01010	FAOut, SUB
SUBI	01110	FAOut, SUB
CMPI	01111	FAOut, SUB
SUCI	01101	FAOut, SUB, *UseC*
SUC	01100	FAOut, SUB, *UseC*
NOP	11000	ShOut
'F'	11001	ShOut
NEG	11010	FAOut, SUB, ZeroA
'D'	11110	FAOut
LSL	11111	ShOut, ShL, {Sh8-1}
LSR	11101	ShOut, ShR, {Sh8-1}
ASR	11100	ShOut, ShR, *ShSign*, {Sh8-1}
AND	10000	AND
OR	10001	OR
XOR	10011	XOR
NOT	10010	NOT
NAND	10110	NAND
NOR	10111	NOR
LLI	10101	ShOut, LLI
LUI	10100	ShOut, ShL, ShB, Sh8
	11011	-

SUB inverts the second input and flags for a subtraction operation and *ZeroA* sets the first input to zero. *ShL* and *ShR* switch between left and right shifting while *ShB* switches between using the first (A) or second (B) input to shift. Signals *Sh8*, *Sh4*, *Sh2* and *Sh1* enable each section of the barrel shifter and during normal shifting operations are dependent upon the 4 bit immediate input to decoder. *Sh8* is set during a **LUI** instruction since it will always shift by 8 bits. Signal *LLI* activates the LLI module after no operation is performed within the main ALU. While *UseC* and *ShSign* are internal signals indicating use of the carry flag from the previous instruction and shifting in the current sign bit respectively. The remaining signals enable the relevant tristate buffers to output the result. It uses one hot encoding to prevent contention. The one unused Opcode does not activate any signals to reduce the number of gates within the decoder.

$$\text{FAOut} = \bar{A} + BD\bar{E} \tag{3.1}$$
$$\text{SUB} = \bar{A}BC + \bar{A}B\bar{C}E + B\bar{C}D\bar{E} + \bar{A}CDE \tag{3.2}$$
$$\text{ShL} = ABCDE + A\bar{B}C\bar{D}\bar{E} \tag{3.3}$$
$$\text{Sh8} = (ABCE + ABC\bar{D})imm[3] + A\bar{B}C\bar{D}\bar{E} \tag{3.4}$$
$$\text{Sh4} = (ABCE + ABC\bar{D})imm[2] \tag{3.5}$$
$$\text{Sh2} = (ABCE + ABC\bar{D})imm[1] \tag{3.6}$$
$$\text{Sh1} = (ABCE + ABC\bar{D})imm[0] \tag{3.7}$$
$$\text{ShOut} = AC\bar{D} + ABCE + AB\bar{C}\bar{D} \tag{3.8}$$
$$\text{UseC} = \bar{A}C\bar{D} \tag{3.9}$$
$$\text{ShR} = ABC\bar{D} \tag{3.10}$$

By using the Opcodes and K-map groupings defined in Table 2.2, logic equations have been formed as shown in Equations (3.1) to (3.10). Where the letters A-E correspond to bits 4-0 of the Opcode. Refining these equations for implementation considered the set of logic gates available within the library, as well as the number of gates needed. Since the negated output gates could have more inputs, as well as being physically smaller, they were more favourable to use. Some potential options for the logic equation for the *SUB* signal are shown in Equations (3.11) and (3.12). The first one, taken from the K-map, requires 9 non-inverting logic gates. This equation cannot be implemented since `AND` gates with more than 2 inputs are not available. Equation (3.12) requires 8 gates and also cannot be implemented using the available library. However if inverted using DeMorgan's law to produce Equation (3.13) it is possible. Since no further improvements could be made, this final equation is used and implemented in Figure 3.8. A similar approach was used for the remaining signals, also shown in Figure 3.8.

The additional logic needed for the flag calculations is implemented in the decoder. However, the circuit diagram for this portion is shown in Figure 3.8. For the control signals which respond to only one Opcode, a gate array was used, as shown in Figure 3.9.

$$\text{SUB} = \bar{A}BC + \bar{A}B\bar{C}E + B\bar{C}D\bar{E} + \bar{A}CDE \tag{3.11}$$
$$= \bar{A}B(C + \bar{C}E) + D(B\bar{C}\bar{E} + \bar{A}CE) \tag{3.12}$$
$$= \overline{\overline{(\bar{A}B\overline{(\bar{C}\overline{(\bar{C}E)})})}\,\overline{(D\overline{(\overline{(B\bar{C}\bar{E})}\,\overline{(\bar{A}CE)})})}} \tag{3.13}$$

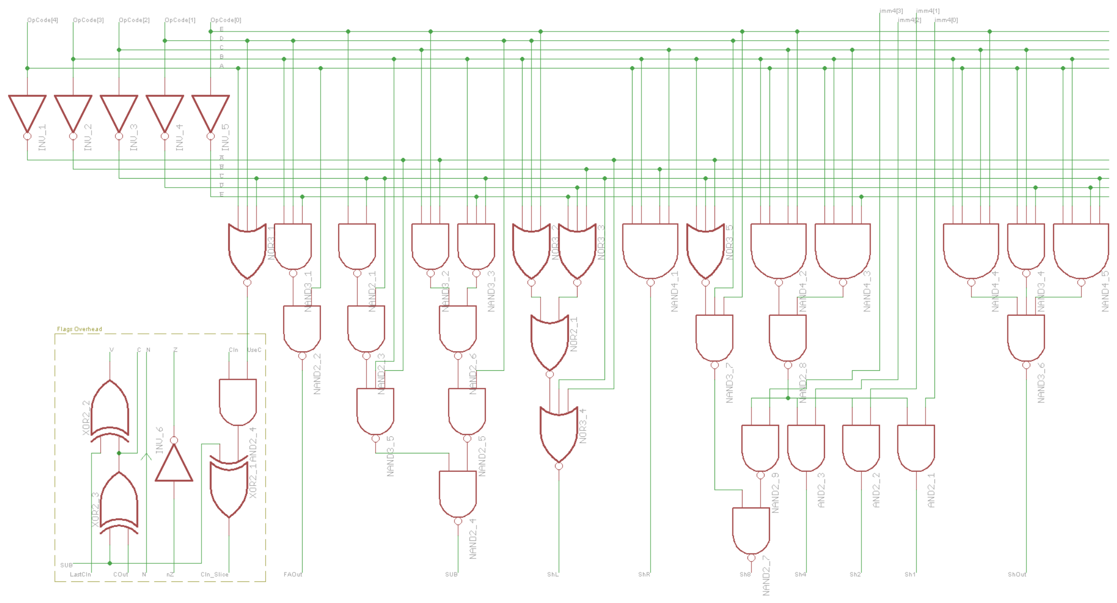


Figure 3.8: Circuit Diagrams For Signals Active For More Than One Opcode

3.4.3 ALU Block

The final hierarchical view of the assembled ALU made up of each part mentioned previously is shown in Figure 3.10. The ALU slice is duplicated to make up the 16 bits in parallel, LLI high is added to the top half and LLI low is added to the bottom half. The decoder is added above, with additional wiring to connect together right shifting inputs. The left shifting inputs at the bottom are tied to ground. The output to the ALU is available as a direct connection to the rest of the datapath, but it is also stored in a register for later transfer onto the system bus. This register forms a part of a separate module to the ALU.

3.5 Datapath

This covers the design of the whole datapath, including circuit diagram. The datapath consists of arrays of submodules. The interrupt constant is also included in the datapath design. The main submodule is the datapath bit slice.

The program counter, link register, general purpose registers and ALU slices are grouped together into one bitslice. This module also implements multiplexors

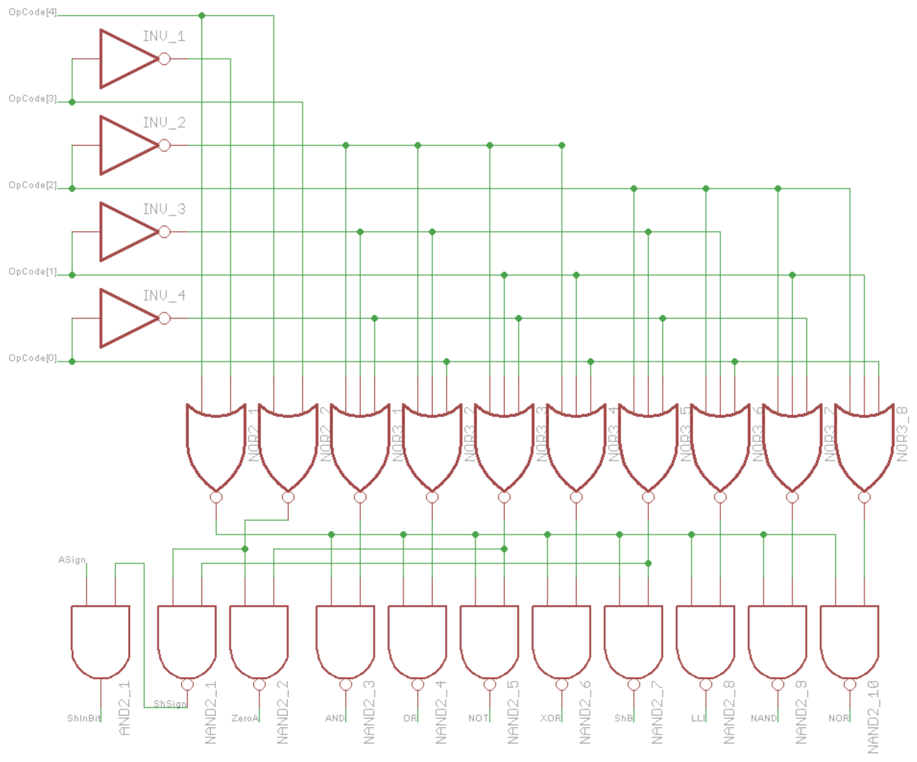


Figure 3.9: Circuit Diagrams For Gate Array and Flag Overhead Logic

for the Write Data selection and the ALU operand selection. Other signals are extended to the edges or routed to the relevant locations.

A "seventeenth" slice is also made. This includes the three decoders for the registers and the decoder for the ALU. The register selection multiplexors are implemented in this module as well. An ALU override was also implemented in this slice. A series of multiplexors are added to override standard ALU behaviour.This override was added after the initial design to be able to increment / decrement the stack pointer irrespective of the current instruction. This was needed to implement the interrupt support to be able to save the Program Counter to the stack.

The final module needed is for the end of the datapath. This has the right cell buffer, along with the ALU output register and a tristate buffer. It also contains an extra tristate buffer to connect the status register (from the controller) to the system bus to allow the storage of the flags. There are two versions of this module: one to connect an input signal to the system bus, and one to drive the remaining

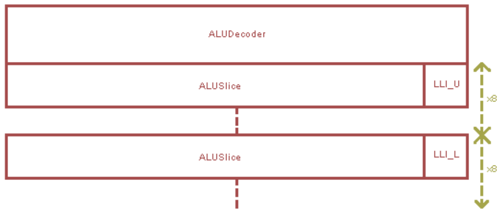


Figure 3.10: Modular Diagram of Assembled ALU

12 bits of the system bus with a 0. This module also contains the memory enable tristate buffer.

The full datapath is made up of the following modules:

1. Datapath Slice

2. Instruction Register modules

3. Left cell buffer

4. LLI cells

5. Right end module

6. Slice seventeen decoders

All blocks are connected in the main datapath module. The instruction register is routed to the decoders at the top of the module and to the bottom to connect to the controller. The system bus outputs and inputs are routed to the right hand side of the module to connect to the pad ring. Control signals are made available at the bottom of the design to connect to the controller.

3.6 Controller

The controller is a Mealy machine used to sequence operations performed on the datapath. Figure 3.11 contains all inputs, outputs and internal registers. There are 7 labelled inputs including two 4 bit buses and one 8 bit bus. There are 26 labelled outputs including five 2 bit buses and one 3 bit bus. Typedefs, defined in a global file, are used for some single bit outputs and all bus outputs to keep decoding consistent in control and the datapath. Inputs from the datapath also use typedefs as these are defined by the instruction set.

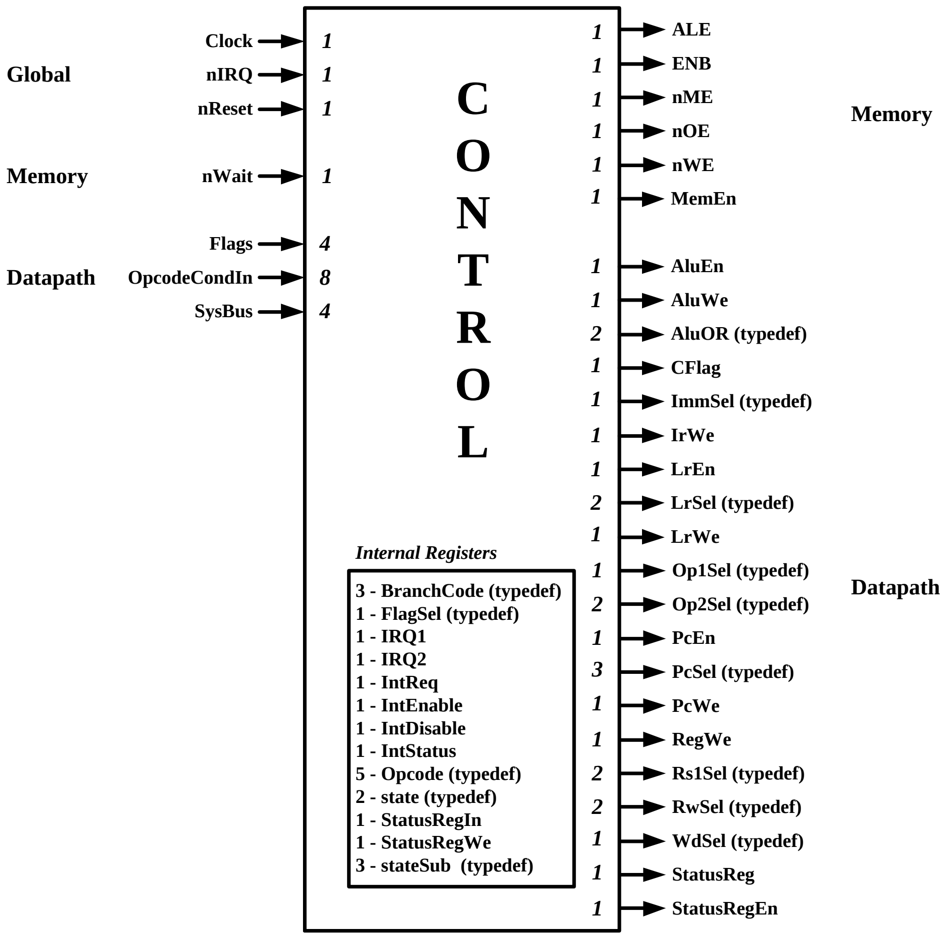


Figure 3.11: Inputs, outputs and internal registers of the control block.

Three main states exist to service the fetch, execute and interrupt stages. Five sub states are used within the main states to further coordinate operation. The substate binary values are gray coded to avoid glitches in memory access. The ASM chart in Figure 3.12 describes state changes using the sub state, current opcode and interrupt request signal. State machines linked to the main state in Figures 3.13 and 3.14 described the substate transitions. The pruned Verilog in Listing 3.1 explains how nested case statements are used to describes large amounts of logical operations. When *nReset* is asserted the machine will asynchronously return to the first cycle of the fetch state.

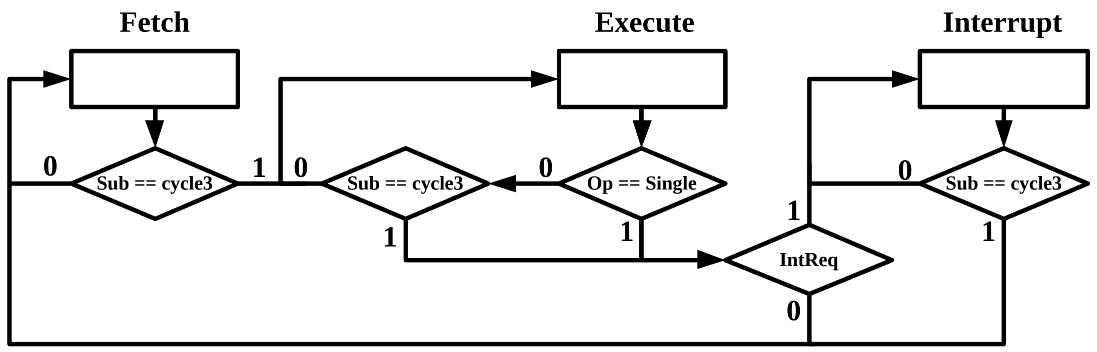


Figure 3.12: ASM chart of controller main states.

Listing 3.1: Reduced Verilog of control unit combinatorial logic.

```
always_comb begin
  outputs = default_outputs;
  case(state)
    fetch:
      case(stateSub) ... endcase
    execute:
      case(stateSub)
        cycle4:
          case(Opcode)
            OPS: ...
            BRANCH:   case(BranchCode)  ... endcase
            INTERRUPT:  case(BranchCode)  ... endcase
          endcase
        cycle0: case(Opcode) OPS: ...   endcase
        cycle1: case(Opcode) OPS: ...   endcase
        cycle2:
          case(Opcode)
            OPS: ...
            INTERRUPT: case(BranchCode) ... endcase
          endcase
        cycle3:
          case(Opcode)
            OPS: ...
            INTERRUPT: case(BranchCode) ... endcase
          endcase
    interrupt:
      case(stateSub) ... endcase
  endcase
end
```

3.6.1 Fetch

Fetching an instruction from memory requires placing the program counter on the address bus then reading the instruction in to the instruction register. The state machine described in Figure 3.13 asserts signals associated with memory access and delays if the slow memory access input, *nWait*, is set. When the transition from **cycle3** to **cycle0** is made the main state machine, Figure 3.12, also makes a transition therefore severing the link to this machine.

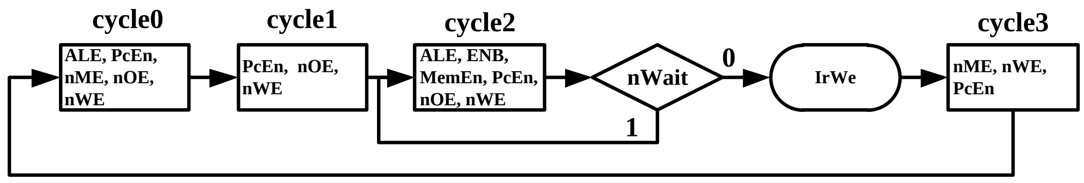


Figure 3.13: ASM chart of sub state transitions in the fetch stage.

3.6.2 Execute

The instruction fetched from memory is decoded and executed in either 1 or 4 cycles. This is a variable length stage dependant upon whether the instruction requires memory access. After the instruction has been executed the *IntReq* signal is checked to decide whether an interrupt has occurred. If an interrupt has occurred the substate on exit is **cycle4** otherwise it is **cycle0** ready to perform another fetch.

A large section of Listing 3.1 contains control flow for output decoding in this stage. The branch code is used during branch operations and also interrupt operations.

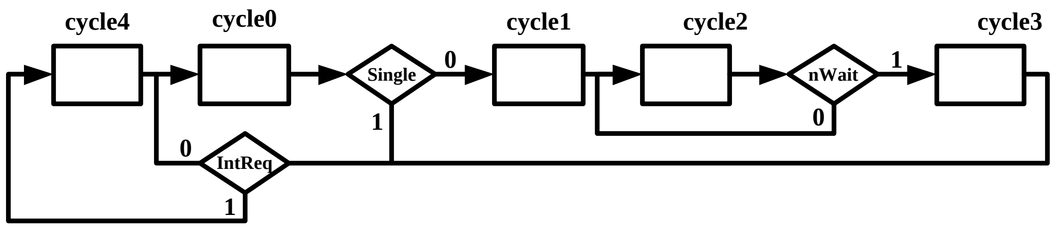


Figure 3.14: ASM chart of sub state transitions in the execute stage.

3.6.3 Interrupt

The circuit in Figure 3.15 is used to generate an interrupt request. To avoid problems introduced by metastability the external *nIRQ* signal is retimed using two D-types. *IntEnable* can only be set in code using the **ENAI** instruction. *IntDisable* is set when the processor is interrupted and also in code using the **DISI** instruction. The end register contains *IntStatus* which is low at reset therefore interrupts are off by default.

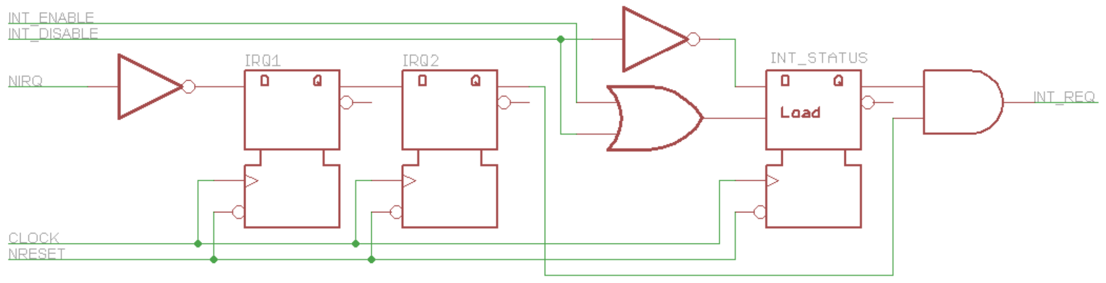


Figure 3.15: IntReq generation circuit.

Entering an interrupt service routine (ISR) is completely handled in hardware. Interrupts are disabled and override functionality is used to decrement the stack pointer therefore allowing the program counter to be pushed to the stack. The program counter is forced to address $0x0010$ which contains the code for ISR. The first operation in the ISR is **STF** which places the flags, located in the control unit, on the stack.

The penultimate operation in the ISR should be **LDF**. This loads the flags back in to the control unit. This is similar to a **POP** operation but writes to the control unit instead of a general purpose register. The final operation is a **RETI** which again is similar to a **POP**, but writes to the program counter.

A bidirectional data interface to the control unit is required to facilitate these operations. Figure 3.16 contains the circuit used to load and store flags inside the control unit. The 4 bit word containing the flags can be sourced from either the ALU or the bottom bits of *SysBus*. When storing the flags in memory the output of the status register must be placed on *SysBus* where the upper 12 bits are all set to zero.

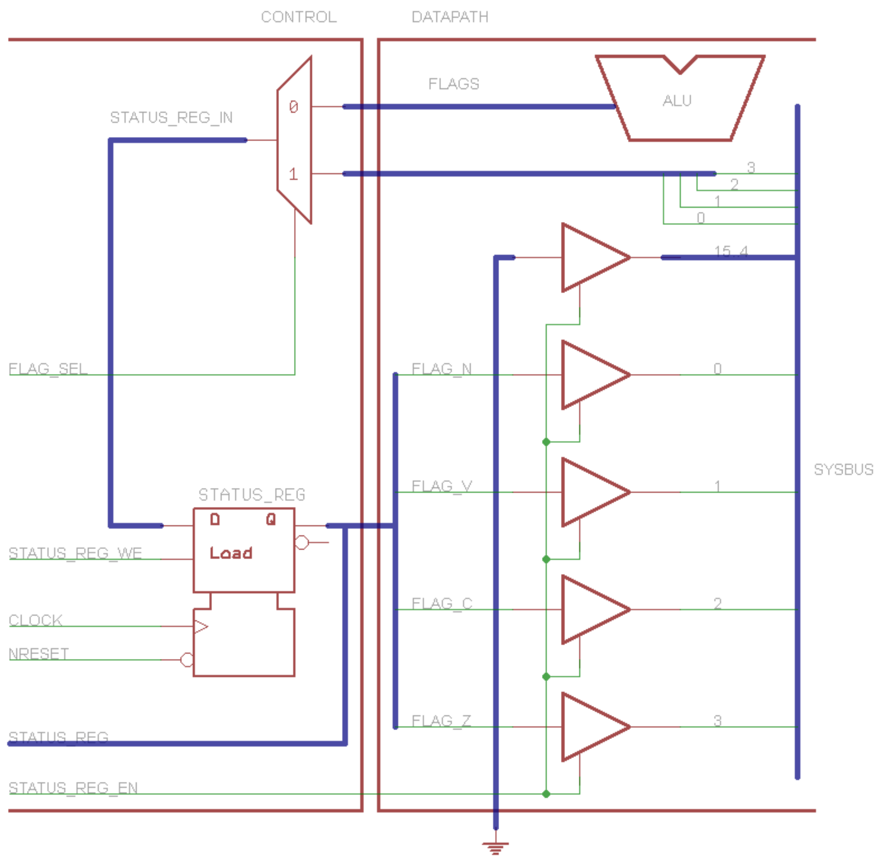


Figure 3.16: Control unit flag interface.

3.6.4 Synthesis

The controller was synthesised using Cadence Encounter. This provided a gate level netlist which was then used for place and route. Both Magic and L-Edit place and route tools were used.

The Magic place and route algorithm had a considerably larger routing channel than L-Edit algorithm as the cells were all placed in one long row. This was suboptimal as it resulted in a lot of dead space within the floorplan.

The L-Edit place and route provided far better results. The datapath and controller were approximately the same width. The I/Os were able to be positioned around the module to reduce the overall wiring needed in the full floor plan. Most of the datapath control signals were routed to the top of the module to reduce the wiring channel needed between the controller and datapath. All the memory access signals were routed to the left of the controller to make easier connections to the pad ring. The controller was also connected to the low nibble of the input data. This was connected at the bottom of the controller to reduce wiring needed. Finally, decoder signals and flag inputs were routed to the right of the control.

3.7 CPU

The datapath and the control units were designed to allow them to connect together with as small a wiring channel as possible. All the I/Os were placed in each of the blocks to minimize the routing between the module and the pad ring. The pad ring used had a core size of $1505.00\mu m \times 1500.70\mu m$. The total size of the chip including the pad ring is $2150.00\mu m \times 2145.70\mu m$, a total area of $4.6mm^2$.

The controller was positioned below the datapath as it needed a connection to the lowest nibble of the system bus. The control signals between the datapath and controller were then placed to minimise the routing between them. This placement is at the expense of some control signals being routed to above the datapath.

A net list was created and the magic autorouter was used. This routed all the signals between the controller, datapath and the pad ring. The power lines were routed manually to the correct sizes.

A large amount of polysilicon and the name of the processor, SAMURAI, is placed in the routing channels of the CPU to satisfy the design rule checks. A N-ohmic guard ring is also placed around the design and is connected to the core supply. A large block of not_pwell paint is used to explicitly define all undefined areas.

4. Testing

4.1 Register Block

Tests were implemented for the decoder and bitslice modules implemented in Magic. A full module test was also written. The same test was designed to be run on both the Magic layout and the behavioural SystemVerilog design. This allowed the full operation of the behavioural model and implemented layout to be verified to the same standard.

The bitslice test loads one bit of data into each register, and checks that it can be read back on each of the data read lines. Assertions are used to automate the checking of the bit slice.

The decoder is verified by exercising all the possible inputs to each of the decoders. The one hot encoding output is then checked to be as expected for the current input. This is repeated for each read decoder, and the write decoder is verified to only output correctly when the write enable signal is high.

The full test uses a “task” code block to write to a register and read the data back. The data is then verified to be correct on both read ports. This process is repeated 100 times to exercise different data on all the registers. Assertions are used to give a final pass/fail result allowing quick verification of the module. The tests ensure the read buffers and the writing to each register is fully functional.

4.2 Program Counter

This module contains both the program counter and link register, which could be tested independently by checking the response to different select signals. The PC was tested by setting *PcSel* to each possible value, then causing a change in the value of that input to ensure the value of PC reflects the change. This covers incrementing, *SysBus*, *AluOut* and constant inputs.

The link register was then tested in the same manner for loading from *SysBus* and an incremented program counter value. The value of each register was read from the system bus with assertions to ensure correct behaviour.

4.3 Instruction Register

The test for the instruction register is for the magic layout only, as it is not an explicit module within the behavioural model. This test is written to exercise the storing and reading of the instruction register, and the sign extension.

The test begins by resetting the module and checking the instruction register output is 0. The register is then loaded with 65535 and verifies the output matches this value.

The sign extension is then tested. This is done by loading a value which should be sign extended if it is a 5 bit immediate, but not if it is a 8 bit immediate. The two outputs are verified for each value of the *ImmSel* signal. This is then repeated by using a value that would be extended if it is 8 bits, but not if it is 5 bits.

This verifies the operation of the instruction register and that the sign extension functions correctly. All behaviour is verified using assertions with a final pass/fail statement which gives the number of errors if any are present. This allowed for modifications to be easily verified.

4.4 Arithmetic Logic Unit

To promote the use of hierarchy within design the ALU is broken down into a number of sub-modules. These have been tested individually to ensure they operate as expected. Then they are combined into 16 bits without decoder for testing the combination of ALU and LLI slices. Since the LLI slices are either a single multiplexor or wires they will not be tested individually. They will form a part of the ALU Block testing.

4.4.1 ALU Slice

Testing of this module was broken down into the different ALU functions to be performed: arithmetic, logic and shifting. Control inputs which effect the behaviour of arithmetic operations are *SUB* and *ZeroA*. As such tests were run to cover every combination of *A*, *B* and *CIn* for addition and subtraction. Then *ZeroA* was activated during each combination of *B* and *CIn*, while *A* is held high, to ensure correct tying low of one input. For subtraction tests, because additional logic for handling subtraction carrys is within the decoder, testing needed to account for both the carry in and borrow signals being the inverse of expected operation. Logic operations are tested by using every input combination possible and comparing to the relevant logic table expected. Testing of the shifting capabilities was done by setting all inter-slice inputs high with the input A held low. Then for each case of: left, right and left with B input shifting, different immediate signals were set to observe the propagation of bits through the slice. If a bit in the 4 bit immediate is high, a 1 would be outputted in the next adjacent bit of the output. This is *OutLeft* for left shifting or *OutRight* for right shifting. Although shifting operation was as expected, much clearer testing will be done during 16 bit block tests to emulate typical running.

4.4.2 ALU Decoder

The main testing of the ALU decoder module requires ensuring a correct response to each possible Opcode. This was done by cycling through in-putted Opcode values, with assertion checking using an always statement which activates every time an input changes. These check against the logic equations for each output signal given in Equations 3.1 to 3.10. As well as Table 3.2 for single Opcode signals. Signals *UseC* and *ShSign* are still checked for correspondence with this table, even though they are not outputs from the decoder module.

Flag testing sets the Opcode to either **ADC** or **SUC** and checks each combination of relevant inputs to ensure V, C and are as expected. N remains untested as there is no logic to test. Shift operations set the 4 bit immediate to a range of values which cover each output Sh8-Sh1 being activated at least once. A full test from 0 to 15 is not needed. The input bit for arithmetic shifting is also tested by enabling/disabling ASign.

4.4.3 ALU Block

Testing of 16 connected slices took a similar approach to an individual slice, but with full 16 bit values. Arithmetic used two arbitrary input values combined with each possible carry in value, accounting for both addition and subtraction operation. Logic tests ensure each bit pair is operated upon independently of the rest by using an arbitrary string of bits. Whilst shift tests run through both directions and a number of shifting amounts, similar to slice testing, to show the selected input is shifted by the expected number of bits. Then the **LUI** instruction was tested separately to ensure successful 8 bit left shifting operation. Finally **LLI** was tested using the same values as in previous tests, to check upper and lower byte concatenation of inputs A and B respectively.

4.5 Datapath

Testing of the complete datapath was done at both the slice level and fully assembled. These modules are highly hierarchical and as such each sub-module would have been tested during the implementation of it. Since the datapath slice is simply an accumulation of sub-modules and some multiplexors, only the top-level circuitry needs to be tested. This was made up of input selection multiplexors for regBlock and the ALU. The first was tested by changing the inputs *AluOut* and *SysBus* and switching between them, asserting if the output is correct. A similar approach was taken for the remaining logic.

Prior to testing it was necessary to reset register values, which was done by extending *Clock* and *nReset* signals to the edge of the cell as testing inputs. This does break hierarchy rules within magic, but prevents the need for a slightly larger cell to access global signals.

The assembled datapath was tested by checking the flow of data between each node on the diagram shown in Figure 1.1, with additional multiplexed nodes between the Instruction Register and register address select signals; *Rs1*, *Rs2* and *Rw*. Each of these nodes is an array of internal signals either from each slice or as outputted from the decoders in slice 17. Then these nodes are grouped in accordance with their location in the datapath and tested separately. These groupings were: multiplexors from Instruction Register, Operand Selection, ALU output, link register connections, program counter connections, writing general purpose registers, ALU override and status register. As with slice testing, every operation performed by the ALU or GPRs were not tested as this is covered by the individual module test.

4.6 Controller

State machine activity is recreated in the testbench which is verified using observable control unit outputs. This assumes a known state on reset and state transitions are done by manipulating the inputs as described in Section 3.6. The test then applies to both the behavioural model and the netlist extracted from synthesis. An umbrella test is used to verify no output of the controller is ever unknown during any state. A series of test vectors are applied to the module, an example is described in Listing 4.1.

Listing 4.1: Control unit test vector.

```
DoFetch(LSL);
@(posedge Clock);
SyncOutput(0,0,0,nOR,0,ImmShort,0,0,LrSys,0,0,1,0,0,Op1Rd1,Op2Imm,1,
    Pc1,1,1,Rs1Ra,RwRd,WdAlu,0);
state = fetch;
stateSub = cycle0;
$display("LSL - OK");
```

Activity in the control unit is dependent on the upper byte of data in the instruction register which is broken down into the 5 bit Opcode and the 3 bit branch condition code. The instruction register is populated in the fetch stage, which is always the same for control signals, so a function `DoFetch()` is used to update the instruction register with a different value for testing. Inside the function the substate transitions are made ending with **cycle0** on exit. Memory access violations are tested using the assertions applied in the **demux_bus.sv** file along with the entire system as a larger framework is required.

After each fetch the variable length execute stage is tested. The output is verified using the function `SyncOutput()`. This probes the datapath as such to verify if the control unit is performing the required operations. State transitions occur outside the function the test will end if the outputs are ill-matched.

As the interrupt functionality is a lower priority no explicit testing of entry was performed at this level. This proved difficult because of the nature of real-time systems. Operations used to support the interrupts can be tested using the existing method. Actual interrupt functionality is tested when running programs on the entire CPU.

4.7 CPU

A full system white noise test was devised. This consisted of filling the program memory with random data. Two simulations are then run: one with the behavioural model and one on the extracted netlist. If the two models are identical, then the contents of the memory and the registers should match at the end of the simulations.

A few issues were encountered in the implementation of the test. Firstly, memory operations caused issues. As the memory map contains undefined areas, load operations could possibly load invalid memory into the design. Due to the Von-Neuman architecture, this invalid memory could be stored back to the program memory and cause the processor to crash. Invalid memory cannot be verified, resulting in the test failing. For this reason, all memory operations were removed from the test data. Similarly, jumps were also removed to prevent the program jumping to an invalid memory location.

Interrupt instructions were also removed due to the memory returning issue with **RETI**, **STF**, **LDF**, and the lack of verification for the enable and disable interrupt instructions. Finally, due to the unpredictable nature for some flags after logic operations, any arithmetic instructions with flag dependencies (**ADCI**, **ADCIB** etc.) are also removed. The removed operations are replaced with **ADDIB R0 #127** to save repeatedly checking the data.

The Program Counter is probed depending on which simulation is being run, and checked on the rising edge of each clock pulse. If the Program Counter is greater than or equal to $0x07FF$ (the end of valid memory), the simulation is terminated. The values of the registers are saved to a file.
A python script was written to do the following:

1. Generate 2048 words of random data
2. Remove any invalid instructions (as described above)
3. Write operations to a hex file
4. Run Behavioural simulation
5. Run Extracted simulation
6. Compare the saved files

The comparison checks the final register values and indicates a pass or fail. A pass is issued if all registers match. The test on the processor showed that there is an inconsistency between the two processors. The output of the test is shown in Listing 4.2. However, due to the use of random data, this is not a repeatable test and it is difficult to debug the issues.

Listing 4.2: Output of the white noise test

```
Reg   B      E      P/F
------------------------
0     007f   007f   P
1     5504   0073   F
2     faff   faff   P
3     581a   581a   P
4     aafc   aafc   P
```

```
5       04d4     04d4     P
6       7cff     0001     F
7       faf4     faf4     P
White Noise Test Failed.
```

5. Conclusion

The processor designed is a 16 bit RISC architecture which has a fully completed layout and is design rule checked. With added novel functionality including; multi-bit shifting, stack-specific operation `PUSH` and `POP`, varied length immediate value support and 8 GPRs with dedicated link register and program counter. Each module has been individually tested and verified. However, there is a discrepancy between the behavioural model and the extracted layout. In the individual modular test sequences, this discrepancy was not noticed. It is not known if the bug exists in the layout or the behavioural model.

Appendix B discusses the workings of the group and the project management. This is rounded off with individual reflections from group members. Appendix C shows the overall Division of Labour within the group for the entire project.

Given the size of the group, the level of functionality gained and the progress made is higher than the original expectations of the group. However this has lead to a less optimized design due to the limited manpower available. As with interrupt support being an additional aspect not forming a part of the initial design. It also became apparent from the "Balloon Debate" that our design focused more on advanced functionality than operating speed and physical size which should have had greater consideration.

Overall, the project has been successful in producing a functional general purpose microprocessor. Whilst supporting material such as a symbolic assembler has been produced to a good standard to aid any programmer with using this system.

A. Instruction Set Summary

	Mnemonic	Syntax	Semantics	Flags		Opcode	Cond
1	ADD	ADD Rd, Ra, Rb	Rd ← Ra + Rb	c,v,n,z	A	00010	-
2	ADDI	ADDI Rd, Ra, #imm5	Rd ← Ra + imm5	c,v,n,z	A	00110	-
3	ADDIB	ADDIB Rd, #imm8	Rd ← Rd + imm8	c,v,n,z	B	00011	-
4	ADC	ADC Rd, Ra, Rb	Rd ← Ra + Rb + c	c,v,n,z	A	00100	-
5	ADCI	ADCI Rd, Ra, #imm5	Rd ← Ra + imm5 + c	c,v,n,z	A	00101	-
6	NEG	NEG Rd, Ra	Rd ← 0 - Ra	c,v,n,z	A	11010	-
7	SUB	SUB Rd, Ra, Rb	Rd ← Ra - Rb	c,v,n,z	A	01010	-
8	SUBI	SUBI Rd, Ra, #imm5	Rd ← Ra - imm5	c,v,n,z	A	01110	-
9	SUBIB	SUBIB Rd, #imm8	Rd ← Rd - imm8	c,v,n,z	B	01011	-
10	SUC	SUC Rd, Ra, Rb	Rd ← Ra - Rb - c	c,v,n,z	A	01100	-
11	SUCI	SUCI Rd, Ra, #imm5	Rd ← Ra - imm5 - c	c,v,n,z	A	01101	-
12	CMP	CMP Ra, Rb	Rd ← Ra - Rb	c,v,n,z	A	00111	-
13	CMPI	CMPI Ra, #imm5	Rd ← Ra - imm5	c,v,n,z	A	01111	-
14	AND	AND Rd, Ra, Rb	Rd ← Ra `AND` Rb	n,z	A	10000	-
15	OR	OR Rd, Ra, Rb	Rd ← Ra `OR` Rb	n,z	A	10001	-
16	XOR	XOR Rd, Ra, Rb	Rd ← Ra `XOR` Rb	n,z	A	10011	-
17	NOT	NOT Rd, Ra	Rd ← `NOT` Ra	n,z	A	10010	-
18	NAND	NAND Rd, Ra, Rb	Rd ← Ra `NAND` Rb	n,z	A	10110	-
19	NOR	NOR Rd, Ra, Rb	Rd ← Ra `NOR` Rb	n,z	A	10111	-
20	LSL	LSL Rd, Ra, #imm4	Rd ← Ra << imm4	n,z	A	11111	-
21	LSR	LSR Rd, Ra, #imm4	Rd ← Ra >> imm4	n,z	A	11101	-
22	ASR	ASR Rd, Ra, #imm4	Rd ← Ra >>> imm4	n,z	A	11100	-
23	LDW	LDW Rd, [Ra, #imm5]	Rd ← Mem[Ra + imm5]	-	C	00000	-
24	STW	STW Rd, [Ra, #imm5]	Mem[Ra + imm5] ← Rd	-	C	01000	-
25	LUI	LUI Rd, #imm8	Rd ← imm8, 0	-	B	10100	-
26	LLI	LLI Rd, #imm8	Rd ← Rd[15:8], imm8	-	B	10101	-
27	BR	BR LABEL	PC ← PC + imm8	-	D	-	000
28	BNE	BNE LABEL	(z==0)?PC ← PC + imm8	-	D	-	110
29	BE	BE LABEL	(z==1)?PC ← PC + imm8	-	D	-	111
30	BLT	BLT LABEL	(n&∼v `OR` ∼n&v)?PC ← PC + imm8	-	D	-	100
31	BGE	BGE LABEL	(n&v `OR` ∼n&∼v)?PC ← PC + imm8	-	D	-	101
32	BWL	BWL LABEL	LR ← PC + 1; PC ← PC + imm8	-	D	-	011
33	RET	RET	PC ← LR	-	D	-	010
34	JMP	JMP Ra, #imm5	PC ← Ra + imm5	-	D	-	001
35	PUSH	PUSH Ra	R7 ← R7 - 1; Mem[R7] ← Ra;	-	E	-	-
		PUSH LR	R7 ← R7 - 1; Mem[R7] ← RL;				
36	POP	POP Ra	Mem[R7] ← Ra; R7 ← R7 + 1;	-	E	-	-
		POP LR	Mem[R7] ← RL; R7 ← R7 + 1;				
37	RETI	RETI	PC ← Mem[R7]	-	F	-	000
38	ENAI	ENAI	IntEnFlag ← 1	-	F	-	001
39	DISI	DISI	IntEnFlag ← 0	-	F	-	010
40	STF	STF	R7 ← R7 - 1; Mem[R7] ← Flags;	-	F	-	011
41	LDF	LDF	Flags ← Mem[R7]; R7 ← R7 + 1;	c,v,n,z	F	-	100

B. Project Management

Groups members are referred to by initials in this Chapter as follows:

1. HL - Henry Lovett (hl13g10)
2. MW - Martin Wearn (mw20g10)
3. AJR - Ashley Robinson (ajr2g10)
4. ARR - Anusha Reddy (arr1g13)

B.1 Group formation

The group initially contained five students. At the start of the project, two of the original students left the group: one joined another team, and one dropped the module. This left the group with three members - AJR, MW and HL. Discussions were then undertaken as to if the group wanted / needed a fourth member. As the group could not come to a unanimous decision, one for a fourth member, one against and one who was happy either way, Iain's advice for having a fourth member was taken.

The original member of the group did not wish to rejoin the group. Instead, a colleague of the original member was offered to join.

In the first discussions with the group, HL was appointed as group leader.

B.2 Meetings

The group held two meetings per week at scheduled times: Monday afternoons at 15:00 and Thursday afternoons at the end of the lab. All meetings were chaired by HL.

Agendas for Monday meetings were sent out by HL, usually 24 hours beforehand. The Monday meeting was there to discuss any work done since Thursday or outstanding issues. The other task for the Monday meeting was to confirm the milestones for the week. This ensured that all group members were aware of who was responsible for what that week to avoid confusion that could potentially cause a delay in a hand-in. Ongoing tasks were discussed at these meetings also. Work for the lab on Thursday was allocated so members were able to commence work without hindrance.

The meeting at the end of the Thursday lab was to discuss the progress made that day. As this was the main working time, smaller aspects were discussed during the day between individual members. The meeting at the end of the lab allowed everyone to know any design changes or issues encountered and the problems discussed. Work for the time outside of the lab was then allocated at this point.

B.3 Skills Assessment

A small skills audit was done in the first meeting. The results of this are seen in Table B.1. It was apparent early on that ARR was not comfortable with a lot of the skills required. Realising this early in the project allowed the work to be allocated optimally throughout the group members, particularly ensuring ARR was given realistic objectives and adequate help where required.

Table B.1: Skills Audit for Team R4. A 5 means very competent, 1 is not competent.

Task	HL	MW	AJR	ARR
Verilog	4	4	4	2
Magic Layout	4	4	3	4
Assembly Language	3	3	3	1
Processor Design	4	4	4	1
Git	4	3	4	1

B.4 Work Breakdown

The skills audit provided the basis for the initial work allocations. Preferences were given and the group was roughly broken into two sections: HL and AJR were to pursue the SystemVerilog development, MW and ARR would do the Magic layout. Although the work was distributed this way, it was by no means a comprehensive divide. ARR and MW were to also write SystemVerilog testbenches, and HL and AJR would be responsible for the synthesis and layout of the control unit. The initial breakdown was done to give the team a starting point.

B.5 Git

HL, MW and AJR all had worked together previously, therefore the use of Git revision control was decided before the group was formed. When ARR joined the group, our file control was discussed, and alternative options were discussed. One idea from ARR was to use DropBox. This was a simple to use system, but would cause issues when multiple people worked on the same files. There was also no log of changes and broken files could have accidentally been integrated. The full recursive revision control with Git helped as if and when a file was committed that didn't work, it could be reverted to the previous version. Change logs are also used with Git, allowing to see which files have changed, by who and why.

The advantage to using Git was that the work could be branched. This was exceptionally useful as the group was looking to implement interrupts. The interrupt work could be done concurrently, but separately, to the normal work flow. If the group hadn't achieved the milestones for the interrupt development, no time would have been needed removing it. At the point of success with interrupts, the work done was easily merged into the master branch.

All group members agreed on the use of Git. HL and MW both spent time with ARR to help her understand the basics to be able to use Git.

B.6 Github

Github is an online resource for hosting Git repositories. As well as the hosting, Github provides an issue tracking facility. Issues were opened for items of work to be done, or for bugs found in the design. The issue webpages provided a comment thread for the group members for discussion.

Issue tracking meant it was easy to keep an eye on what tasks were on going, complete or if it needed more time and resources to complete on time. Issues can be allocated to group members, so the same task is not completed by multiple people. Github was extremely useful for this tracking and provided a common communication network for the group.

B.6.1 Git Additions by Group Member

Github has the ability to extract statistics of the repository. This includes the commits, additions and deletions over time by each user (excluding merges). The number of commits is not an accurate method of measuring activity as different people have different commit habits. For example, one member may commit a lot, but make little changes in each commit, whereas someone else may make a lot of changes, test and then commit. Deletions are also included in the statistics, but are not a reliable measure as HL was responsible for keeping the repo clean and removing the temporary or unused files.

However, the total additions is a more accurate measure. Magic files can be included in this measure as these are plain text files. Report writing was done using LaTeX, which in essence, is code and therefore the additions measure applies here too.

Due to ARR being introduced to Git, it would be expected that in the initial phase of the project, her commit count would be low, resulting in low additions also. Ideally, this would then increase to a reasonable level during the project. Some commits were done on behalf of ARR by HL in the early stages also, but this was not common.

Figure B.1 shows the addition history in the repository. It is clear to see that HL employs a commit little and often style, where as MW and AJR commit more in less commits. The initial peak with AJR was the addition of an emulator he found. If this is not taken into account, the total additions of HL, AJR and MW are approximately equal. It was expected that the additions of ARR would be lower than other members due to the learning curve needed for Git. However, it is clear to see the 6,225 additions from ARR is significantly less than the rest of the group.

B.7 Reflections

This section contains individual reflections from group members. These are written to give everyone a chance to discuss their view of the group operation. ARR's reflection is not included. She was advised to include her own reflections in her individual report and given the same suggestions as to what it should include.

B.7.1 HL

Firstly, I regard the project as a success. We managed to work together and communicate to produce a functional processor. All members got on well with each other on a whole and no social issues were encountered. As MW, AJR and myself have all worked together previously, which helped going into this project. The regular meetings also helped keep everyone informed. There were no issues wtih who was meant to be handing work in, or what people were doing.

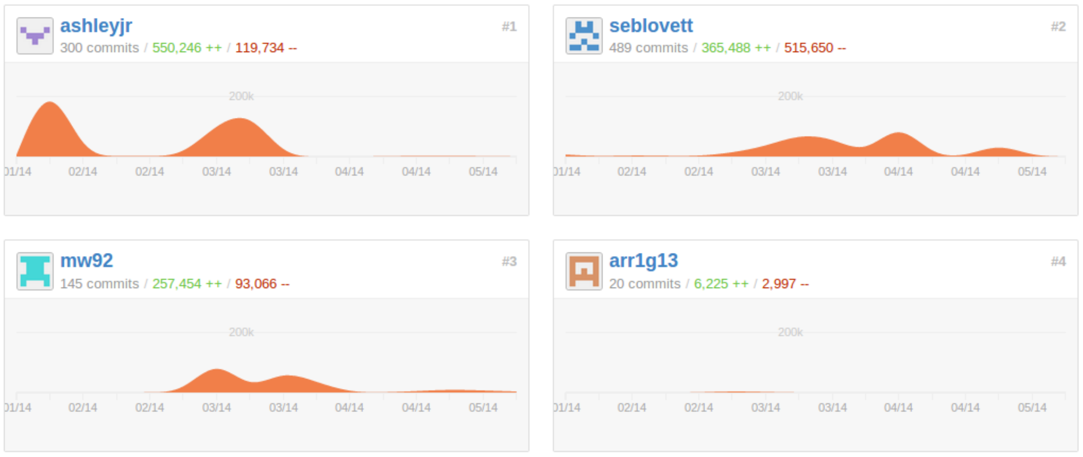


Figure B.1: Graph from Github showing the additions over time for each user. Additions are measured in lines in a file and shown in green. Deletions are shown in red. (seblovett = HL; ashleyjr = AJR; mw92 = MW; arr1g13 = ARR)

I think it was difficult for ARR to join our group as MW, AJR and myself were all good friends. There was a skill and motivation divide which I noticed early on in the project. Initially, work was allocated evenly amongst the group. Work from ARR was often of a lower standard than other members. In the early stages, help was given to ARR by other members. However, as the project progressed with no change, less important work was given to ARR due to the time pressures and necessity to fix work done.

In hindsight, the pursuit of the interrupts was probably too much for a small group. I was cautious to allow development of this at the time, but agreed to give it an attempt. AJR and myself conducted the work needed. As we got it functional, and were the only team to do so, this encouraged us to keep the functionality. However, this probably distracted us from other issues in our processor, such as the speed and size. If we had more time, I would have given ARR more work to help her learn, and I potentially was a little impatient as my expectations may have been too high. Nonetheless, I have learnt a lot about project management from doing this project. It's been an enjoyable project and the end result is something I am proud of.

B.7.2 MW

In my opinion the project design proposed and undertaken was realistically acheivable within the time-scale provided for a standard size group of 5 members. The lack of one member would not however create a significant impact. While a team of three I felt would put too much pressure on us to maintain a comparable level to other groups. Due to ARR not being a fourth year student like the rest of us, she was not capable of as much. While she also appeared to lack the motivation to learn new skills, and contribute enough to the project. The addition of interrupts was also implemented in an ad-hoc manner. Which with a dedicated role would have enabled a much more seamless design to be implemented. Overall the project progressed well but could have benefited more with greater manpower to spread the workload and allow for more design optimisations.

B.7.3 AJR

All four members of the team were capable of making an equal contribution to this project. HL took a management position educating himself in all areas of the design to enable coordination between myself and the datapath experts. MW and ARR were given the task of building the datapath using the layout tools provided where MW was more the capable and decided to investigate producing an assembler which proved beneficial. ARR found layout and simple circuit design in her strengths but some of the designs she produced were of poor quality and required re-working by either HL or MW. Her knowledge of Verilog was poor and simple verification tasks, required for the first semester course, were beyond her ability.

The management tool Git was only used by myself and HL at the beginning of the project. Our strong recommendation lead to it being used throughout where the remaining two members, with our assistance, would have to learn how to use the tool. MW did not struggle but ARR did and was reluctant to make an effort. In retrospect this could be viewed as detrimental but with the benefits a tool like this provides I do not think this is the case.

Our main downfall was reaching too far with limited human resources. Operating a smaller team but going beyond the compulsory scope of this project left room for optimisation in basic areas. However producing the functioning deliverables we have is a true testament to our skills in both digital design and project management.

C. Division of Labour

	Task	Percentage Effort on task			
	ECSID:	hl13g10	ajr2g10	mw20g10	arr1g13
1	Initial Design	25	25	25	25
2	Verilog Behavioural Model	50	50	0	0
2.1	Interrupts	40	40	20	0
3	Multiply Program	0	100	0	0
4	Magic Datapath	20	0	60	20
5	Verilog Cross Simulation	50	50	0	0
6	Control Unit Synthesis	100	0	0	0
7	L-Edit Control Unit	100	0	0	0
8	Final Floorplanning, Placement and Routing	100	0	0	0
9	Factorial Program	0	100	0	0
10	Random Program	0	100	0	0
11	Interrupt Program	0	100	0	0
11	Verilog Final Simulations and Cadence DRC	100	0	0	0
12	Assembler	2.5	2.5	95	0
13	Programmer's Guide Documentation	30	20	40	10
14	Final Report	35	25	40	N/A
14.1	Introduction	100	0	0	N/A
14.2	Architecture	100	0	0	N/A
14.3	Instruction Set	0	0	100	N/A
14.4	Implementation	33	33	34	N/A
14.5	Testing	33	34	33	N/A
14.6	Conclusion	60	0	40	N/A
14.7	Project Management	100	0	0	N/A
	OVERALL EFFORT	32	32	31	5