Final Report
ELEC6027: VLSI Design Project

Format title page

Team R4
Henry Lovett (hl13g10)
Ashley Robinson (ajr2g10)
Martin Wearn (mw20g10)
Anusha Reddy (arr1g13)

30th April, 2014

Todo list

Contents

Chapter 1

Introduction

INCOMPLETE CHAPTER: Introduction

Overview of the report

Chapter 2

Architecture

INCOMPLETE CHAPTER: Architecture

Design of the datapath architecture.
Refer to the research done and how this influenced the design
Incl. diagram

Chapter 3

Instruction Set

INCOMPLETE CHAPTER: Instruction Set

Design of instruction set (Started)

Refer to research (yes, but not sourced)

ISA novelties (some integrated, no dedicated section)

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. This gave rise to a RISC based architecture since they have a small number of instructions and are optimized for a smaller chip area. They also promote a simpler datapath since the same length instructions can be bit-sliced into more identical slices. Irregular lengths will 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.

Expand basic ISA design considerations

Since a 16-bit microprocessor was to be designed, it was decided to base the system on the ARM Thumb architecture. This is a subset of the main 32-bit ARM instruction set which contains a suitably complete set of instructions. However it included a number of operations which take advantage of the ARM's 32-bit datapath, so any high register operations were removed. Change of state, sign extension and debugging instructions, among others, were also removed for simplification or because they were not necessary. This produced the original ISA made up of instructions 1-4, 6-10, 12-16, 20-24, 27-32, 35 and 36 as noted in the summary table in Appendix A. While in-

structions 5 and 11 were added to support use of carry flag with an immediate value. 17, 18 and 19 are included to form a complete logic set. 25 and 26 are for loading an initial value to any general purpose register. Instruction 33 is included from the SPARC ISA for returning from a procedure. Instruction 34 enables a control jump to anywhere in 2^{16} memory locations. While instructions 37-41 were added for support of a single interrupt.

Within this ISA it was decided to support up to 3 operands. This allows greater flexibility with the instructions available and reduces the amount of memory required to perform data processing operations. 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 referenced in the remaining 10 bits. With the option of expanding the third operand to a 5 bit immediate value. This benefits from how common it is that a small immediate value is used more than a larger one. 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 if required. However RISC systems have a lower density than CISC, because the latter is capable of operations such as automatic context saving within the same instruction for performing a procedure call. The density of this system has been improved by using 3 operand instructions to reduce the number of data transfers required. This is illustrated in Figure 3.1 in terms of register transfers required to add two register values and place the result in another register.

explain multiple shifting support

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

Figure 3.1: Comparison of Operand Amounts

Both control transfer and interrupt operations, with one exception, do not use any registers. This meant the free space could be used for further definitions, leaving only one opcode being needed for each. 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. However since there is a limitation on the distance, it was deemed necessary to include the instruction "JMP" which takes a register address for transferring to any position in memory.

	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**												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

Table 3.1: General Instruction Formatting

The concept of orthogonality in instruction formatting involves the separation of bits into different fields which can each be assigned a value independently. Where each field defines a different aspect of the instruction. To promote orthogonality, the instruction formatting for data manipulation operations followed a similar structure to the ARM Thumb, as shown in Table 3.1. Which was adapted to create all other types of formatting, 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 in the instruction. 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. With the resultant

mapping shown in Table 3.2 and Table 3.3, with the important groupings highlighting. The four groupings shown correspond to command signals from decoder to ALU which need to be active for more than one instruction.

Possibly combine Opcode kmap tables into subfigures?

	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	SUBI	LSR	LLI
100	ADC	SUC	ASR	LUI

• FAOut
• ShOut

Table 3.2: Opcode Assignment K-Map A

	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	SUBI	LSR	LLI
100	ADC	SUC	ASR	LUI

• SUB
• ShR

Table 3.3: Opcode Assignment K-Map B

Allocation of Condition codes for control transfer instructions was based upon the type and action of each branch. The aspects considered were: conditional or unconditional, use of link register and flags used. These are summarised in Table 3.4a. 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 un-

conditional 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. As shown in Table 3.4b.

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

(a) Cond. Assignment

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

(b) ICond. Assignment

Table 3.4: Condition Code Assignments

Chapter 4

Design and Implementation

INCOMPLETE CHAPTER: Implementation

4.1 Register Block

Design of whole module, including circuit diagram
- Use of hierarchy / blocks - i.e. bit sliced, decoder
- Design of slice,
- Design of decoder,
- Design of block,
- Layout in silicon

4.2 Program Counter

Design of whole module, including circuit diagram
- Use of hierarchy / blocks - i.e. bit sliced, decoder
- Design of slice,
- Design of decoder,
- Design of block,
- Layout in silicon

4.3 Instruction Register

Design of whole module, including circuit diagram

Use of hierarchy / blocks - i.e. bit sliced, decoder
Design of slice,
Design of decoder,
Design of block,
Layout in silicon

4.4 Arithmetic Logic Unit

Design of whole module - ongoing
Use of hierarchy - implied mention, todo
Design of slice - initial text done
Design of decoder - initial text done
Design of block - initial test done
Layout in silicon

The arithmetic logic unit (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 range of functions needed fall into one of four types: arithmetic, logic, shifting and load lower. With the last type being a special case for the LLI instruction. Arithmetic operations are centralized around a full adder with additional gates for subtraction, setting the first input to zero and flag calculations. The logic unit consists of one gate corresponding to each of the full set of logic instructions supported. While shifts are performed using a barrel shifter to support up to 15-bits 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 4.1, with the output of each section connected to an internal bus through a number of tri-state gates.

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 bits of the ALU will each need a different routing to each segment, reducing how much of each slice could be identically duplicated.

Figure 4.1: ALU Separated into Sub-modules

Initial design of the ALU module was based upon the planned datapath layout and the input/output connections it showed, as seen in Figure 4.2. From this it was seen that there may be some difficulty with interpreting the opcodes provided. Since 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. As such, it was decided to use a separate decoder to interpret the opcodes and send control lines to the ALU for the specific functions to be performed.

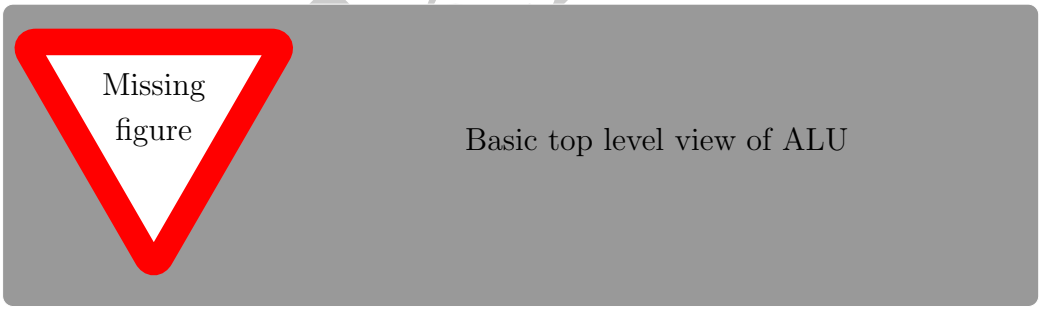

Figure 4.2: Initial Top-level View of ALU Module

4.4.1 ALU Slice

The most beneficial bit slice would be one that performs all the functions required for one bit and as such can be replicated to produce a 16-bit datapath. As such design focussed upon this idea, with considerations towards

the overall size of the ALU once built. The arithmetic section was simple to bitslice since the full adder and input selection gates would be duplicated for each bit anyway. While the flags required only one OR gate for the Z flag and the sum, carry in and carry out signals to be available at the top of the slice. Since a decoder is positioned at the top, additional gates needed for flag calculation can be implemented once within the decoder. This is shown in Figure 4.3. Bit slicing the logic section was just one of each logic gate followed by a tri-state buffer in the slice, shown in Figure 4.4.

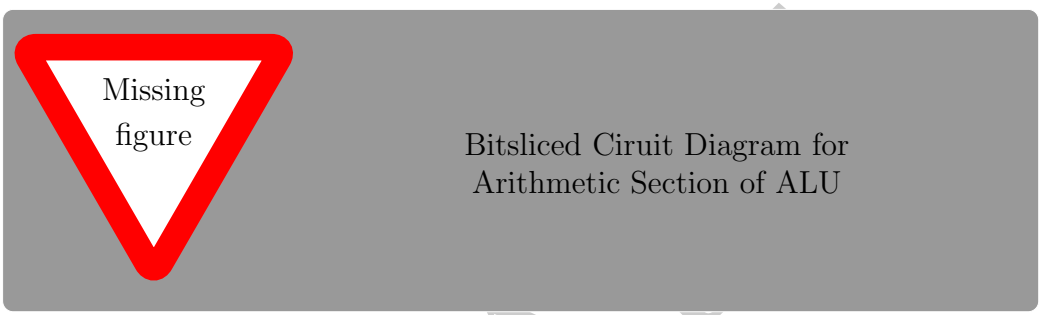

Figure 4.3: Bitsliced Ciruit Diagram for Arithmetic Section of ALU

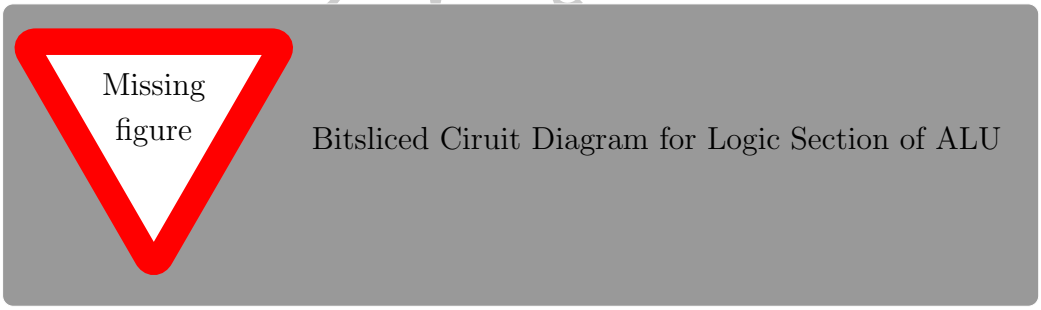

Figure 4.4: Bitsliced Ciruit Diagram for Logic Section of ALU

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 from this slice section having more dependency on the neighbouring slices than both arithmetic and logic functions. Since there is no obvious way of integrating them into the same circuitry, left and right shifting have separate hardware. With the ability to select between

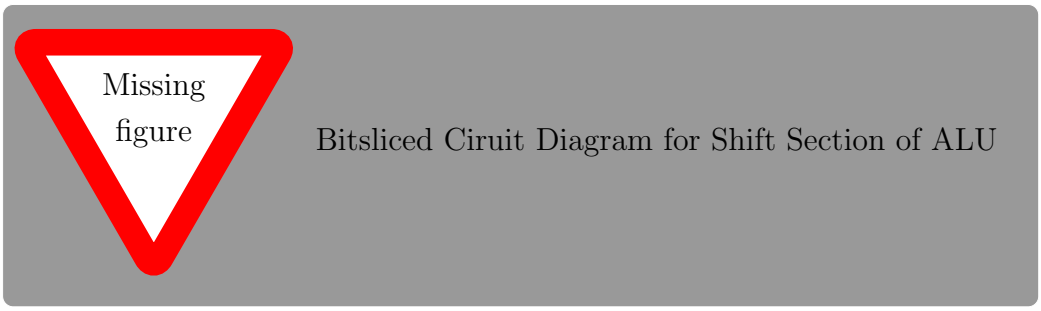

Figure 4.5: Bitsliced Ciruit Diagram for Shift Section of ALU

a zero and the current operand's sign to shift into a right shift operation. Implementation of this section is shown in figure 4.5.

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 4.6. Loading a lower immediate involves concatenating the existing high byte of the destination register with the value in the instruction. However there is no way of separating 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, as shown in Figure 4.7, therefore LLI functionality is not part of the main ALU.

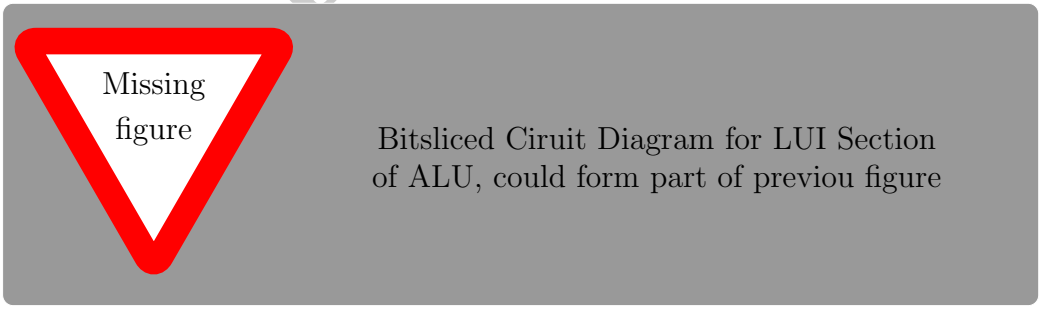

Figure 4.6: Bitsliced Ciruit Diagram for LUI Section of ALU

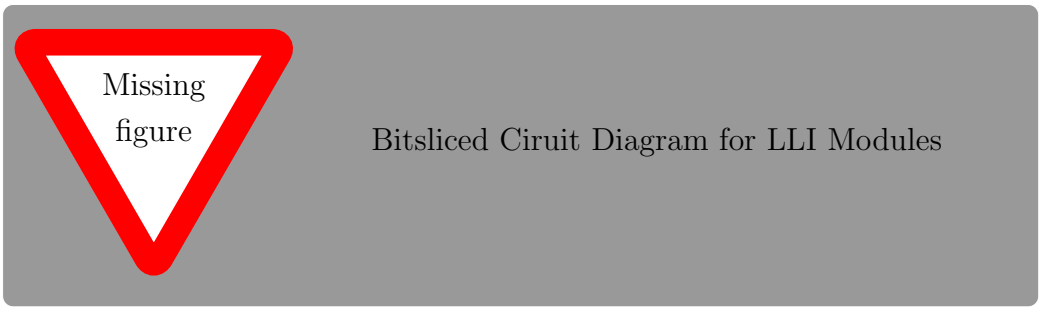

Figure 4.7: Bitsliced Ciruit Diagram for LLI Modules

4.4.2 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 multiple circuits of purely combinational logic. Table 4.1 shows the necessary control signals for each mnemonic.

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

Table 4.1: Control Outputs For Each Available Instruction Mnemonic

check what 11011 does

The signal FAOut enables the tri-state buffer of the arithmetic section. SUB inverts the second input and flags for a subtraction operation and ZeroA sets the first input to zero. The logic tri-states are controlled by the signals

AND, OR, XOR, NOT, NAND and NOR for the relevant logic operations. The signal ShOut enables the tri-state for the shifting section and if used with nothing else active has the effect of no operation being performed on data. 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 always 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 main ALU. While UseC and ShInBit are internal signals indicating use of the carry flag from previous instruction and the bit to shift in respectively. The one unused opcode is set to perform no operation on data to prevent unpredictable behaviour.

$$\text{FAOut} = \bar{A} + BD\bar{E} \tag{4.1}$$
$$\text{SUB} = \bar{A}BC + \bar{A}B\bar{C}E + B\bar{C}D\bar{E} + \bar{A}CDE \tag{4.2}$$
$$\text{ShOut} = AC\bar{D} + ABCE + AB\bar{C}\bar{D} \tag{4.3}$$
$$\text{ShR} = AC\bar{D}\bar{E} + ABC\bar{D} \tag{4.4}$$
$$\text{UseC} = \bar{A}C\bar{D} \tag{4.5}$$
$$\text{Sh1} = (ABCE + ABC\bar{D})imm[0] \tag{4.6}$$
$$\text{Sh2} = (ABCE + ABC\bar{D})imm[1] \tag{4.7}$$
$$\text{Sh4} = (ABCE + ABC\bar{D})imm[2] \tag{4.8}$$
$$\text{Sh8} = (ABCE + ABC\bar{D})imm[3] + A\bar{B}C\bar{D}\bar{E} \tag{4.9}$$

check against implimented

By using the opcodes and k-map groupings defined in Tables 3.2 and 3.3, logic equations have be formed as shown in Equations 4.1 to 4.9. 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. The possible gates essential to the decoder were: and2, or2, nand2, nand3, nand4, nor2 and nor3. Where each number corresponds to the amount of logic inputs. Since the negated output gates had more possible 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 4.10 and 4.11. The first

one, taken from the k-map, requires 1 and3, 4 and4's, 1 or4 and 3 inverters. To simplify design, inverters are not considered towards optimum design as both inverted and non-inverted inputs will be made available globally. This equation is not possible to implement due to AND gates with more than 2 inputs. Equation 4.11 requires 2 and2's, 3 and3's, and 3 or2's which again cannot be implemented. However if inverted using DeMorgan's law to produce Equation 4.12 it is possible, and uses 8 smaller gates. Since no further improvements could be made, this final equation is used and implemented in Figure 4.8. A similar approach is taken for the remaining signals listed in Equations 4.1 to 4.9, with the final circuit diagrams also shown in Figure 4.8.

$$\text{SUB} = \bar{A}BC + \bar{A}B\bar{C}E + B\bar{C}D\bar{E} + \bar{A}CDE \tag{4.10}$$

$$= \bar{A}B(C + \bar{C}E) + D(B\bar{C}\bar{E} + \bar{A}CE) \tag{4.11}$$

$$= A + \bar{B} + \overline{(\bar{C}\overline{(C + \bar{E})})} + \overline{\overline{(\bar{B} + C + E)}(A + \bar{C} + \bar{E})} + \bar{D} \tag{4.12}$$

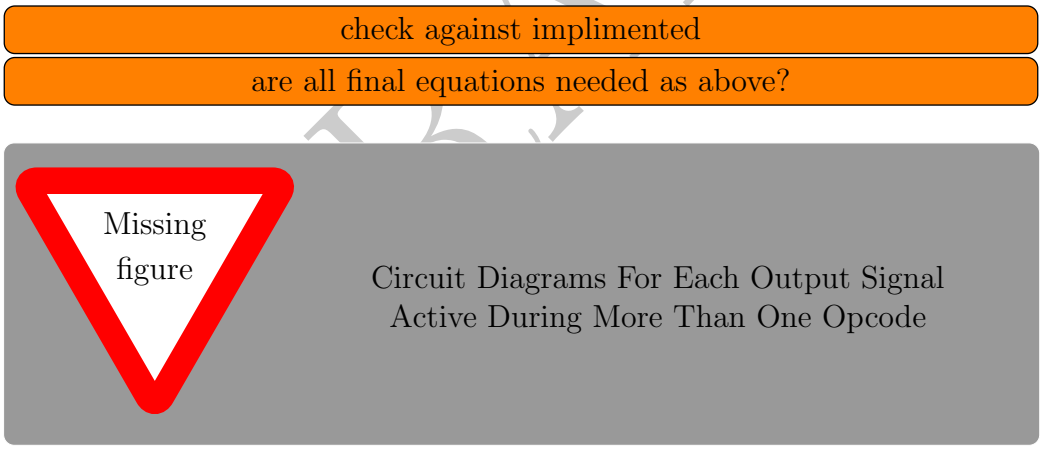

Figure 4.8: Circuit Diagrams For Signals Active For More Than One Opcode

For the control signals which respond to only one opcode, a gate array was used, as shown in Figure 4.9. Since additional logic for flag calculations are implemented in the decoder, the circuit diagram for this portion is also shown in Figure 4.9.

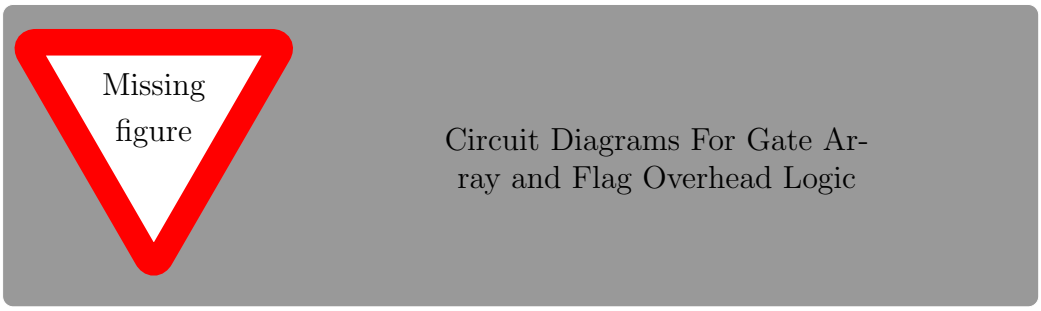

Figure 4.9: Circuit Diagrams For Gate Array and Flag Overhead Logic

4.4.3 ALU Block

The final hierarchical view of the assembled ALU made up of each part mentioned previously is shown in figure 4.10. ALU slice is duplicated to make up the 16 bits in parallel, LLI high is added to the top byte and LLI low is added to the bottom half. The decoder is added above, with additional wiring to connect together right shifting inputs. While the left shifting inputs at the bottom are tied low. The output to the ALU is available as a direct connection to elsewhere in 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.

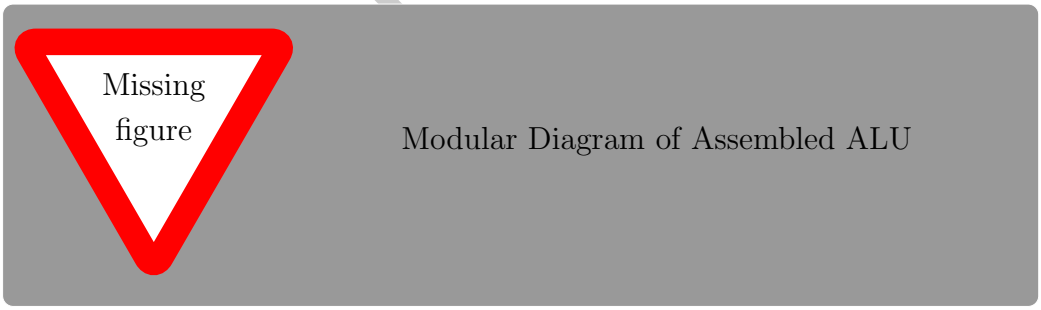

Figure 4.10: Modular Diagram of Assembled ALU

4.5 Datapath

Design of whole module, including circuit diagram

Use of hierarchy / blocks - i.e. bit sliced, decoder
Design of slice,
Design of decoder (slice 17),
Design of block,
Layout in silicon

4.6 Controller

Design of - simple statemachine?
Control signals - description, use of type defs?
Description of main states:
Fetch
Execute
Interrupt
Implementation of interrupts (flags, enable...)
Synthesis and layout - I/O config, magic vs Ledit maybe?

4.7 CPU

Overall layout
pad ring size
positioning of control and datapath
power routing
anything else?

Chapter 5

Testing

INCOMPLETE CHAPTER: Testing

5.1 Register Block

Include Sub tests - of slice and decoder (if app)
Explain tests - what is done
why it is done.
How it verifies everything - why it is complete
Show simulation results

5.2 Program Counter

Include Sub tests - of slice and decoder (if app)
Explain tests - what is done
why it is done.
How it verifies everything - why it is complete
Show simulation results

5.3 Instruction Register

Include Sub tests - of slice and decoder (if app)
Explain tests - what is done
why it is done.

How it verifies everything - why it is complete
Show simulation results

5.4 Arithmetic Logic Unit

Include Sub-module tests

Explain tests - what is done

why it is done

How it verifies everything - why it is complete

Show simulation results

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 a 16-bit ALU without decoder for testing the combination of ALU and LLI slices. As well as more useful shifting tests.

5.4.1 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, subtraction and both with ZeroA enabled. The test results are shown in Figure 5.1. For subtraction tests, because additional logic for handling subtraction carrys is within the decoder, the outputted response shows inverted Cin and Borrow signals. For both of the last two groupings input A is active during the length of testing, which shows that zero is used when ZeroA is active. Whilst testing logic operations, each gate is tested for its full range of input combinations to show correct operation. The output of these tests are shown in Figure 5.1. Testing of the shifting capabilities involved 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 controlling the amount to shift is high, a 1 would be outputted in the next adjacent bit of the output. This is OutLeft for left shifting of OutRight for right shifting as shown in Figure 5.1.

check explaination

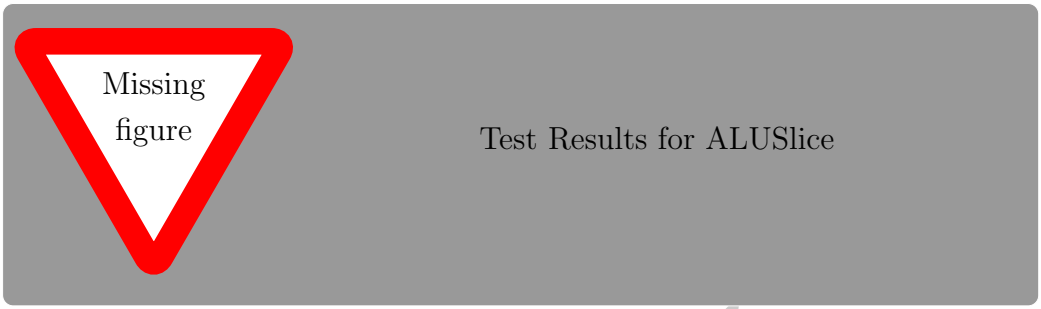

Figure 5.1: Test Results for ALUSlice

Adjust shifting output to include aluout

5.4.2 ALU Decoder

The main testing of the ALU decoder module requires ensuring a correct response to each possible opcode. Secondary testing covers the flag calculation circuitry and the bit to input for right shifting operations. The test results are shown in Figure 5.2, where each line indicates what input changed and which control and flag outputs are active as a result.

Basic tests run through each opcode and ensure the outputted signals are as expected from Table 4.1. The signal *CIn_Slice* is the carry input to the top ALU slice after CIn has been influenced by UseC and SUB. While the signal *ShiftIn* is the bit used for right shifting accounting for the current operation and data sign. The second block of tests are for carry checking, while the third is for shift bit checking. With a final test for correct negation of nZ to Z.

check

Moves test start names, possible clarity issue?

5.4.3 ALU Block

5.5 Datapath

Include Sub tests - of slice and decoder (if app)

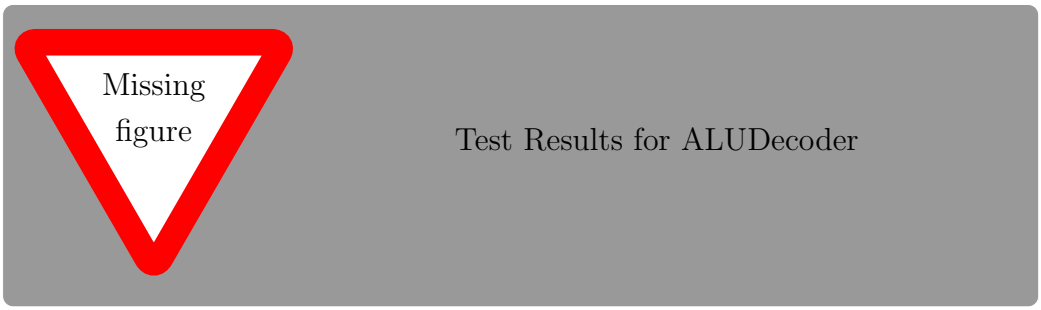

Figure 5.2: Test Results for ALUDecoder

Explain tests - what is done
why it is done.
How it verifies everything - why it is complete
Show simulation results

5.6 Controller

Include Sub tests - of slice and decoder (if app)
Explain tests - what is done
why it is done.
How it verifies everything - why it is complete
Show simulation results

5.7 CPU

Include Sub tests - of slice and decoder (if app)
Explain tests - what is done
why it is done.
How it verifies everything - why it is complete
Show simulation results

Chapter 6

Conclusion

INCOMPLETE CHAPTER: Conclusion

Generic concluding marks

Appendix A

Instruction Set Summary

INCOMPLETE CHAPTER: Instruction Set Summary

	Mnemonic	Syntax	Semantics	Flags	Encoding	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 PUSH LR	Mem[R7] ← Ra; R7 ← R7 - 1; Mem[R7] ← RL; R7 ← R7 - 1;	-	E	-	-
36	POP	POP Ra POP LR	R7 ← R7 + 1; Mem[R7] ← Ra; R7 ← R7 + 1; Mem[R7] ← RL;	-	E	-	-
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	Mem[R7] ← Flags; R7 ← R7 - 1;	-	F	-	011
41	LDF	LDF	R7 ← R7 + 1; Mem[R7] ← Flags;	c,v,n,z	F	-	100

Appendix B

Project Management

INCOMPLETE CHAPTER: Project Management

Use of git
regular meetings

Appendix C

Division of Labour

	Task	Percentage Effort on task			
	ECSID:	hl13g10	ajr2g10	mw20g10	arr1g13
1	Initial Design	100	0	0	0
2	Verilog Behavioural Model	100	0	0	0
3	Multiply Program	100	0	0	0
4	Magic Datapath	100	0	0	0
4.1	Registers	100	0	0	0
4.2	Program Counter	100	0	0	0
4.3	Instruction Register	100	0	0	0
4.4	ALU	100	0	0	0
5	Verilog Cross Simulation	100	0	0	0
6	Control Unit Synthesis	100	0	0	0
7	Magic Control Unit	100	0	0	0
8	Final Floorplanning, Placement and Routing	100	0	0	0
9	Factorial Program	100	0	0	0
10	Random Program	100	0	0	0
11	Interrupt Program	100	0	0	0
11	Verilog Final Simulations and Cadence DRC	100	0	0	0
12	Assembler	100	0	0	0
13	Programmer's Guide Documentation	100	0	0	0
13.1	Architecture	100	0	0	0
13.2	Assembler	100	0	0	0
13.3	Instruction Set	100	0	0	0
13.4	Programming Tips	100	0	0	0
13.5	Programs	100	0	0	0
13.6	Register Description	100	0	0	0
13.7	Simulation	100	0	0	0
14	Final Report	100	0	0	0
14.1	Introduction	100	0	0	0
14.2	Architecture	100	0	0	0
14.3	Instruction Set	100	0	0	0
14.4	Implementation	100	0	0	0
14.5	Testing	100	0	0	0
14.6	Conclusion	100	0	0	0
14.7	Project Management	100	0	0	0
	OVERALL EFFORT	100	0	0	0