Final Report
ELEC6027: VLSI Design Project

Format title page

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

27th 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 instructions 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.

3 Operands	$R1 \leftarrow R2 + R3$
2 Operands	$R1 \leftarrow R2$ $R1 \leftarrow R1 + R3$
1 Operand	$Acc \leftarrow R2$ $Acc \leftarrow Acc + R3$ $R1 \leftarrow Acc$

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.

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

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

4.5 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 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

5.5 Datapath

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

5.6 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