Computer Systems and -architecture

Extra Exercise: Example Practical Exam

1 Ba INF 2022-2023

Brent van Bladel
brent.vanbladel@uantwerpen.be

In this exercise, we will create a datapath that uses an adress pointer. Try to create everything from scratch without looking at previous projects.

1. Start by implementing a 8-bit program counter. You can use the logisim adder and register.

name	in/out	width	meaning
Branch Relative	I	1 bit	if set, the Branch Offset will be added to the PC as well
Branch Absolute	I	1 bit	if set, the PC will be set to the Branch Offset
Branch Offset	I	8 bit	the Branch Offset
C	I	1 bit	clock input
reset	I	1 bit	if set, the PC is reset to 0
instruction address	O	8 bit	the address of the instruction in the instruction memory

2. Implement an 8-bit ALU. You can use the logisim adder and subtractor.

name	in/out	width	meaning
OP code	I	2 bit	operation code
A	I	8 bit	first input of the operation
B	I	8 bit	second input of the operation
R	O	8 bit	result of the operation

Your ALU should support the following operations:

OP code	operation	meaning
00	OR	R = A OR B
01	ADD	R = A + B
10	SUB	R = A - B
11	INV	R = - A

3. Implement a register file that stores 8-bit data in 8 registers. You can use the logisim register.

name	in/out	width	meaning
rs	I	3 bit	register `rs` index number
rt	I	3 bit	register `rt` index number
rd	I	3 bit	register `rd` index number
Data	I	8 bit	used as input for the write operation, i.e., the new $rd value
C	I	1 bit	clock input
write	I	1 bit	if set, data will be written to rd
reset	I	1 bit	reset all registers?
S	O	8 bit	`$rs`; register `rs` content
T	O	8 bit	`$rt`; register `rt` content
AP	O	8 bit	`$ap`; register `r7` content

Make sure that register R0 always contains 0. We will use register R7 as our address pointer (AP), which will always be send to an additional output. Note that R7 can still be used as a normal register as well: we can write to is by setting the rd-input to 7, and read from it normally by setting the rs-input or rt-input to 7.

4. We will now create a datapath by combining the previous circuits to implement a set of 9-bit instructions. Start by implementing unary instructions, using the following instruction set:

8 7 6	5 4 3	2 1 0	name	instruction	description
`000`	`rs`	`000`	zero	`zero rs`	\$rs := 0
`000`	`rs`	`001`	inc	`inc rs`	\$rs := \$rs + 1
`000`	`rs`	`010`	dec	`dec rs`	\$rs := \$rs - 1
`000`	`rs`	`011`	inv	`inv rs`	\$rs := - \$rs

Already make use of a control unit for these instructions, as this will make it easier to add instructions later. It should have the following inputs and outputs:

name	in/out	width	meaning
Instruction	I	9 bit	the current instruction
rs	O	3 bit	register `rt` index number
rt	O	3 bit	register `rt` index number
rd	O	3 bit	register `rd` index number
regwrite	O	1 bit	if set, data will be written to the register file
ALU OP	O	2 bit	the ALU operation code

Don't forget to add zero constants to the unused program counter inputs!

5. Test your current instructions by executing the following program in logisim:
`inc r1; inv r1; dec r1; zero r1`
You can do so by loading the hexadecimal representation into your instruction memory:
`009 00b 00a 008`

6. Extend you datapath by implementing the following memory instructions:

8 7 6	5 4 3	2 1 0	name	instruction	description
`000`	`rs`	`100`	load	`load rs`	\$rs := MEM[AP]
`000`	`rs`	`101`	store	`store rs`	MEM[AP] := \$rs

You will have to add an 8-bit RAM element to use as data memory. The Address Pointer (R7) always determines where we read or store data in the data memory: you can use the additional AP output of the register file for this purpose.

Also extend your control unit with the necessary outputs:

name	in/out	width	meaning
memwrite	O	1 bit	if set, data will be written to the data memory
memread	O	1 bit	if set, data will be read from the data memory

7. Test your current instructions by executing the following program in logisim:
`inc r7; store r7; load r1`
You can do so by loading the hexadecimal representation into your instruction memory:
`039 03d 00c`

8. Extend you datapath by implementing the following ALU instructions:

8 7 6	5 4 3	2 1 0	name	instruction	description
`100`	`rs`	`rt`	or	`or rs rt`	MEM[AP] := \$rs OR \$rt
`101`	`rs`	`rt`	add	`add rs rt`	MEM[AP] := \$rs + \$rt
`110`	`rs`	`rt`	sub	`sub rs rt`	MEM[AP] := \$rs - \$rt
`111`	`rs`	`000`	inv	`inv rs`	MEM[AP] := -\$rs

Note that these ALU operations write their result directly to the data memory, using r7 (AP) as the address.

9. Test your current instructions by executing the following program in logisim:
`inc r1; add r1 r1; load r2; or r1 r2; load r3; sub r1 r3; load r4; inv r4`
You can do so by loading the hexadecimal representation into your instruction memory:
`009 149 014 10a 01c 18b 024 1e0`

10. Extend you datapath by implementing the following branch instructions:

8	7	6	5	4	3	2	1	0	name	instruction	description
001			offset						brnz	`brnz offset`	if MEM[AP] != 0 then PC = PC + offset + 1
010			address						jump	`j address`	PC = address
011			address						jal	`jal address`	MEM[AP] = PC; PC = address

Also extend your control unit with the necessary outputs:

name	in/out	width	meaning
branch relative	O	1 bit	if set, the PC should perform a relative branch
branch absolute	O	1 bit	if set, the PC should perform a absolute branch
branch value	O	8 bit	the value to be used in a relative or absolute branch

11. Test your current instructions by executing the following program in logisim:
`b 3; inc r1; store r1; j 0; add r1 r1; jal 0`
You can do so by loading the hexadecimal representation into your instruction memory:
`043 009 00d 080 149 0c0`

12. Consider the full instruction set:

8	7	6	5	4	3	2	1	0	name	instruction	description
000			rs			000			zero	`zero rs`	\$rs := 0
000			rs			001			inc	`inc rs`	\$rs := \$rs + 1
000			rs			010			dec	`dec rs`	\$rs := \$rs - 1
000			rs			011			inv	`inv rs`	\$rs := - \$rs
000			rs			100			load	`load rs`	\$rs := MEM[AP]
000			rs			101			store	`store rs`	MEM[AP] := \$rs
001			offset						brnz	`brnz offset`	if MEM[AP] != 0 then PC = PC + offset + 1
010			address						jump	`j address`	PC = address
011			address						jal	`jal address`	MEM[AP] = PC; PC = address
100			rs			rt			or	`or rs rt`	MEM[AP] := \$rs OR \$rt
101			rs			rt			add	`add rs rt`	MEM[AP] := \$rs + \$rt
110			rs			rt			sub	`sub rs rt`	MEM[AP] := \$rs - \$rt
111			rs			000			inv	`inv rs`	MEM[AP] := -\$rs

Imagine we want to extend this instruction set further, how many unused bit-combinations are available for this purpose?

13. Write a program that calculates fibonacci numbers and stores them in sequence in the data memory. Note that you will have to convert your binary instructions to hex manually, in order to load them into Logisim.