Computer Systems and -architecture

Project 2nd Session: Datapath

1 Ba INF 2011-2012

Ruben Van den Bossche
ruben.vandenbossche@ua.ac.be

Sam Verboven
sam.verboven@ua.ac.be

Don't hesitate to contact the teaching assistants of this course. You can reach them in room M.G.2.12 or by e-mail.

Time Schedule

The project for the second session is solved individually. There will be an evaluation during the examination period. At this evaluation moment, you will present your solution of the project by giving a demo and answering some questions.

For this project, you submit a small report of the project you made by filling in `verslag.html` completely. A report typically consists of 500 words and a number of drawings/screenshots. Put all your files in a tgz archive, as explained on the course's website, and submit your report to the assignments on Blackboard.

- Report deadline: **August, 19 2012, 23u55**

Project

1. Build an arithmetic logic unit (ALU) for 16-bit two's complement data words. To do this, create a circuit that implements a 1-bit ALU. Combine them to obtain a 16-bit ALU. Implement the operations below, giving each operation a 4-bit binary code. Your ALU will execute the right operation according to a 4-bit operation input. Next to this, your ALU should have two 16-bit words as input, one 16-bit word as output, and one "error" bit as output, denoting an error.

   Use the Logisim `ALU_GroupXX.circ` file provided on the course page. Your ALU should be able to perform the operations listed below. Make sure to test everything, including the different possible overflow cases!

   (a) **generate 0** (`0000`).
   Example:

result	`0000000000000000`

   (b) **AND** (`0001`).
   Example:

a	`0010010010101010`
b	`1010100101010010`
result	`0010000000000010`

(c) **OR** (0010).
Example:

a	0010010010101010
b	1010100101010010
result	1010110111111010

(d) **NOT** (0011).
Example:

a	0010010010101010
result	1101101101010101

(e) **numeric inverse (two's complement)** (0100).
Example:

a	0010010010101010
result	1101101101010110

Mind overflow!

(f) **numeric addition (two's complement)** (0101). Ripple carry addition suffices.
Example:

a	0010010010101010
b	1010100101010010
result	1100110111111100

Mind overflow!

(g) **numeric subtraction (two's complement)** (0110).
Example:

a	0010010010101010
b	1010100101010010
result	0111101101011000

Mind overflow!

(h) **shift left** (0111).
Example:

a	0010010010101010
result	0100100101010100

(i) **shift right** (1000).
Example:

a	0010010010101010
result	0001001001010101

(j) **signed shift left (two's complement)** (1001). This implements "times two".
Example:

a	0010010010101010
result	0100100101010100

Mind overflow!

(k) **signed shift right (two's complement)** (1010). This implements "divide by two".
Example:

a	0010010010101010
result	0001001001010101

Mind overflow!

(l) **less than** (1011). Results in 1 if $a < b$, 0 if a $\geq$ b.
Example:

a	0010010010101010
b	1010100101010010
result	0000000000000000

(m) **greater than** (`1100`). Results in 1 if $a > b$, 0 if a $\leq$ b.
Example:

a	`0010010010101010`
b	`1010100101010010`
result	`0000000000000001`

(n) **equals** (`1101`). Results in 1 if $a = b$, 0 if a $\neq$ b.
Example:

a	`0010010010101010`
b	`1010100101010010`
result	`0000000000000000`

2. Build a circuit that implements a **16-bit program counter (PC)** that selects an instruction in a RAM element of 20-bit words. Although, from the hardware perspective, instructions of a *not-a-power-of-two* size is inefficient, it suits our demonstrational needs. By default, the PC is increased each clock cycle, and the next instruction is read from memory. You should have the following inputs and outputs:

name	in/out	width	meaning
C	I	1 bit	clock input
instruction address	O	16 bits	the address of the instruction in the instruction memory

3. Build a **register file** made of sixteen 16-bit (Logisim) registers. The register file must be able to read from and write to specified registers. Register 0 is a special case: it always contains zero, and writing to it doesn't modify its contents. The register file has the following in- and outputs:

name	in/out	width	meaning
rs	I	4 bits	register `$rs` index number
rt	I	4 bits	register `$rt` index number
rd	I	4 bits	register `$rd` index number
D	I	16 bits	used as input for the write operation
write	I	1 bit	write to `$rd` enabled?
C	I	1 bit	clock input
S	O	16 bits	register `$rs` content
T	O	16 bits	register `$rt` content

4. In order to translate from the instruction OP-code to the ALU OP-codes and to get all control lines right, you will have to add a **Control Unit** circuit to your datapath.

   - Input is the instruction OP-code (4 bits).
   - Outputs are the ALU OP-code as well as all control lines for i.e. the program counter, instruction and data memory, multiplexers and the register file.

   More information on the implementation of a control unit can be found in Section 4.4 of *Computer organization and design*.

5. Use your register file, your program counter, your control unit, an *instruction* RAM element (16-bit addresses, 20-bit words), a *data* RAM element (16-bit addresses, 16-bit words) and your own ALU to implement a datapath.

19-16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0000	rs				rt				rd				funct				14 R-type Instructions[1]
0001	rs				rt				immediate (unsigned)								lui[3,5]: $rt = imm << 8
0010	rs				rt				immediate (unsigned)								ori[5]: $rt = $rs | imm
0011	rs				rt				immediate (signed[2])								addi: $rt = $rs + imm
0100	rs				rt				immediate (unsigned)								andi: $rt = $rs & imm
0101	rs				rt				immediate (signed[2])								lw: $rt = MEM[$rs+imm]
0110	rs				rt				immediate (signed[2])								sw: MEM[$rs+imm] = $rt
0111	target address																jump: $pc = addr
1000	rs				rt				offset (signed[2])								jr: $pc = $rs+imm
1001	rs				rt				offset (signed[2])								beq: ($rs=$rt) ? $pc=$pc+1+imm
1010	rs				rt				offset (signed[2])								bne: ($rs≠$rt) ? $pc=$pc+1+imm
1011	target address																jal: $ra = $pc + 1;$pc = addr

[1] 14 R-type instructions from your ALU. The ALU opcode is given in the funct field.

[2] Two's complement.

[3] "Load upper immediate": put the 8-bit immediate in the upper 8 bits (shift left x8 and store in register).

[4] The `lui` and `ori` instructions can be used together to implement a `li` pseudo-instruction which loads a 16-bit immediate into a register.

6. Once done, your datapath can correctly execute a program written in machine language, as the behaviour of arithmetic, branching and memory operations is now fully implemented!

7. Exceptions are a very important part of a datapath and control. In this exercise, you will add a basic form of exception handling to your datapath: when an exception is detected, your program counter should halt at the instruction that caused the exception. Both arithmetic overflow and undefined instructions should be detected and supported.

   Think about enhanced versions of exception control. What is necessary in order to add a more advanced form of exception handling to a datapath with our instruction set?

8. Demonstrate the proper operation of your datapath by providing a number of small RASM-programs. Try to use subroutines at least once. Don't forget to initialize the stack pointer. Provide the programs below.

   (a) A program that calculates the Fibonacci numbers and stores them in memory. After which number does overflow occur?

   (b) A program that calculates the greatest common divisor (using the Euclidean algorithm, but without recursion) of two integers read from memory. Store the result back in memory.

   (c) A program that finds the biggest element in an array of integers stored in memory. Store the biggest element back in memory.

   (d) A program that writes a 1 to memory if an array of integers in memory contains duplicates, and writes 0 if it doesn't.

   (e) A program that sorts an array of integers in memory. You can use a sort algorithm of your own choice. If you want a challenge, you can try to implement the quick sort algorithm.