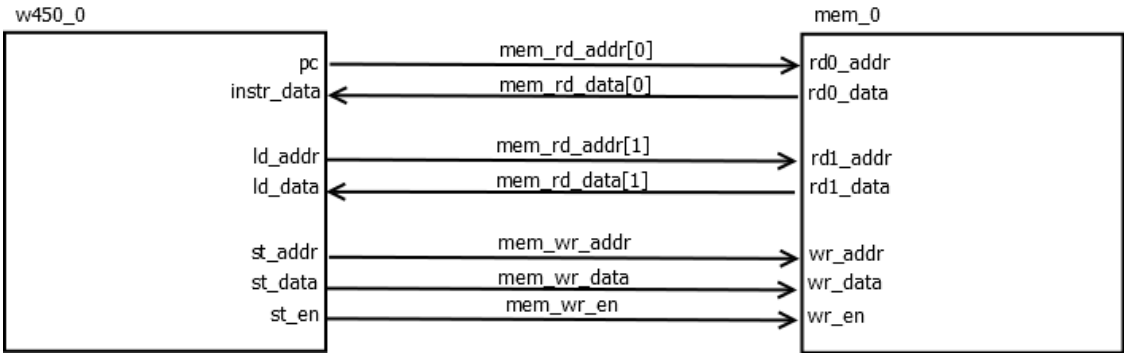
CS450/CS650 Winter 2015 – Assignment 3

Weight 15%
Due Friday, March 20, 11:59pm via submit

1. Implement the w450 multi-cycle processor. Details of the w450 ISA follow. Submit only your `w450.v` file which defines the `w450` module.

   To get started, download the source code in `w450.zip`. It contains the following:
   - `w450.v`: contains a shell of the w450 module for you to complete. It instantiates a `regfile` module that you must use.
   - `regfile.v`: implements a 4-register file with 2 read ports and 1 write port.
   - `mem.v`: contains program instructions and data, populated via a `*.data` file.
   - `w450_tb.v`: contains several testbenches. Each one loads a program from a `*.data` file with a corresponding name. The expected output (timestamps may vary) is enclosed in the comment preceding the testbench. The testbenches (and corresponding marks) are:
     i. tb_fetch [3 marks] Tests fetching of sequential instructions.
     ii. tb_math [3 marks] Tests basic arithmetic operations.
     iii. tb_ldst [3 marks] Tests operations such as `mv, add, sub` that have a source or destination operand in memory.
     iv. tb_branch [3 marks] Tests forwards and backwards branches.
     v. tb_remainder [0 marks] Simple looping program.
     vi. tb_addArray [3 marks] Array summation program.
   - `Makefile`: for use with iverilog. Targets are fetch, math, ldst, branch, remainder, addarray for command-line output only and fetch-debug, math-debug, ldst-debug, branch-debug, remainder-debug, addarray-debug for use with gtkwave.

   The testbench instantiates the `w450` and `mem` modules as soon in this diagram:



w450 ISA

The w450 is an 8-bit processor that has an ISA somewhere between CISC and RISC. Instructions and data reside in the same memory. Word size is 8-bits. Instructions are composed of one or two words, data of one word.

Registers

There are 4 8-bit registers r0-r3.

Addressing Modes

There are 4 addressing modes:

- Register direct – operand comes from register
  e.g. r1
- Register indirect (only for r0) – operand comes from memory at address specified in r0
  e.g. [r0]
- Immediate – 8-bit two's complement data that resides in the word immediately following the instruction in memory
  e.g. #1
- PC relative – byte offset for branch instructions – it resides in the word immediately following the instruction in memory
  e.g. beq r0,r1,-10

Instruction Format

All instructions, except branch instructions, have two operands, rd and rs. Both operands are source operands and the left operand doubles as the destination for the result.
e.g. sub r1,r0 // r1 ← r1 - r0

Branch instructions have three operands, rd, rs and offset. Branch instructions compare the rd and rs operands. The offset is an 8-bit signed number in 2's complement notation.

Instruction Encoding

The 8-bit instruction word is encoded as follows.

7	6	5	4	3	2	1	0
opcode			reg1		reg0		dst

- reg1 and reg0 specify operand registers
  00 = r0, 01 = r1, 10 = r2, 11 = r3
  - r1 – r3 are in direct addressing mode
  - r0 is in direct addressing mode in reg1, and for branch instructions in reg0
  - r0 is in indirect addressing mode in reg0 for add, sub and mv instructions
- dst specifies which of reg1 or reg0 are the destination operand; dst is undefined for branch instructions
- for immediate-mode instructions, addi, subi, and mvi, reg1 is the destination operand, and reg0 and dst are undefined

Instruction Set

add

syntax: add rd,rs
operation: rd ←rd + rs
opcode: 000

add immediate

syntax: addi rd,#data
operation: rd ←rd + data
opcode: 001

sub

syntax: sub rd,rs
operation: rd ←rd - rs
opcode: 010

sub immediate

syntax: subi rd,#data
operation: rd ←rd - data
opcode: 011

move

syntax: mov rd,rs
operation: rd ←rs
opcode: 100

move immediate

syntax: movi rd,#data
operation: rd ← data
opcode: 101

branch if equal

syntax: beq rd,rs,offset
operation: PC ←(rd==rs) ? PC+2+offset : PC+2
opcode: 110

branch if less than

syntax: blt rd,rs,offset
operation: PC ←(rd<rs) ? PC+2+offset : PC+2
opcode: 111

Examples

add r3,r0

operation: r3 ←r3 + r0
encoding: 00000110 (opcode=000,reg1=00,reg0=11,dst=0)
note: r0 must be in reg1 (if it was in reg0, in would be used as a memory address)

add r3,[r0]
operation: r3 ←r3 + [r0]
encoding: 00011001 (opcode=000,reg1=11,reg0=00,dst=1)

add [r0],r3
operation: [r0] ←[r0] + r3
encoding: 00011000 (opcode=000,reg1=11,reg0=00,dst=0)

addi r3,#1
operation: r3 ←r3 + 1
encoding: 00111xxx 00000001 (opcode=001,reg1=11,reg0=xx,dst=x,immediate data=00000001)

blt r3,r0,-10
operation: if r3<r0, PC ← PC+2-10
encoding: 1111100x 11110110 (opcode=111,reg1=11,reg0=00,dst=x,offset=11110110)
offset: -10_{10} = -00001010_2 = 11110110 (2's complement)

Interface (ports)

- st_data [7:0] *output*
  - store data value (data port)
- st_addr [7:0] *output*
  - store data address (data port)
- st_en *output*
  - store data enable
- instr_data [7:0] *input*
  - instruction data value (instruction port)
- pc [7:0] *output*
  - instruction address (instruction port)
- ld_data [7:0] *input*
  - load data value (data port)
- ld_addr[7:0] *output*
  - load data address (data port)
- reset *input*
  - if reset==1, reset PC to 0
  - when reset transitions from 1 to 0, start fetching instruction on next clock cycle
- clk *input*
  - processor clocked on rising edge of clk