cs281: Introduction to Computer Systems

Lab07: Y86 Datapath

1. The purpose of this laboratory is to:
   - develop experience and expertise with the Y86 architecture, and
   - understand how the control inputs need to be asserted for different type of instructions.

2. You must first complete the entire lab on your own. **Once you have the lab complete, you may then check your results with one other student in class (if their lab is complete as well)**. Then the two of you can reconcile any differences you have in your table.

   You will submit a single pdf file of your completed lab chart. Obtain the `chart.tex` file and fill in the entire chart. Submit the LaTeX pdf file of this chart.

3. Obtain the `Datapath.tar` file. Download it. Put it in a clean directory and untar it. You will see the following files in here:

   (a) `Y86.circ` - the main logisim file we will use for this assignment.
   (b) `Y86Memory.circ` - the memory file that goes with Y86.circ.
   (c) `yo2banks` and `yo2banks.c` - A c program that splits up `.yo` files into memory modules that can be loaded into the `Y86.circ` file.
   (d) `irmov.ys` - An example Y86 assembly program with two simple instructions.
   (e) `small.ys` - An example Y86 assembly program with a memory reference.
   (f) `chart.tex` - The blank latex file for your answers.
   (g) `yas32` - The 32 bit version of our assembler.
   (h) `yis32` - The 32 bit version of our simulator.

   Note that this system uses an older 32 bit memory model. Thus we need to use these separate versions of Y86 which are 32 bits.

   This lab will use the `Y86.circ` file. You will also need the `Y86Memory.circ` file which supplements the other circuit file. You will want an original (unmodified) copy of this file to be sure your lab results are correct. You will also need the `yo2banks.c` utility (see below). You may download the latex file for the blank chart from our class Canvas page. All the files for this lab are included in the `Datapath.tar` file.

4. Functional Elements of the Datapath:

   PC : Program Counter – stateful device containing a 32 bit register holding the current program counter. Input is the upcoming PC value, output is the current PC value. A new program counter is stored on the rising clock edge.

IMem : Instruction Memory – stateful device containing the Y86 program, starting at address 0. Input is the 32 bit Program Counter, and output are the 6 consecutive bytes from memory starting at that address. Memory is loaded into four separate banks, each of which is 2 bytes wide, and loading occurs prior to instruction execution. From that point on, the memory may be thought of as combinational, where, based on the PC as input, a six byte output is produced.

ISplit : Instruction Split – combinational circuit whose inputs are the 6 (potential) bytes of the instruction and whose outputs are the union of the fields that make up all the Y86 instructions, including icode, ifun, rA, rB, Dest, and V/D.

Registers : Register File – stateful device containing the eight Y86 registers, numbered from 0 to 7. The register file is capable of routing to output the current values of two of its registers, as well as updating one or two registers. Inputs are srcA and srcB, to determine values sent to the valA and valB outputs, as well as dstE/valE and dstM/valM, which determine register (and its associated value) to write on the next rising clock. If dstE and/or dstM have value 0xF, then no register is updated. Outputs valA and valB always reflect the current value of registers srcA and srcB.

ALU : Arithmetic Logic Unit – combinational circuit capable of performing addl, subl, andl, and xorl. Inputs are ALUfun, ALU_A, and ALU_B. Outputs are ZF, SF, OF, and valE.

CC : Condition Codes – stateful device implemented as a register holding the 3 1-bit values corresponding to ZF, SF, and OF. In addition to these inputs, the register has an enable bit and an asynchronous reset bit as inputs. The output is the current values of the three bits. Updates occur on the rising clock if the enable is also asserted.

DAddr : Data Memory Address – combinational circuit that, given a 32 bit aligned byte address yields a 20 bit word address suitable for input to the data memory. The device also has a single bit indicating whether or not an illegal address was given as its input.

DMem : Data Memory – stateful device used for the read/write data memory involved in a Y86 program. Input is the 20 bit word address A to be read or written along with the 32 bit data value, D, to be modified as a result of a write. Inputs also include a "Store" bit which, when 1, will cause the D value to be stored on the next rising clock edge, as well as an asynchronous reset for the memory. Output is the 32-bit valM corresponding to the value in memory at the word address given by A.

In addition to these functional elements, the datapath consists of a set of multiplexers, one adder unit, and the set of wires connecting all of the elements. The datapath can be made to "execute" instructions by manipulating the control bits, which in this initial datapath are simply input pins. By setting all of these control inputs to particular values based on the current instruction and on other outputs of the functional elements (like the condition codes), we can manipulate the datapath into performing the semantics of our given instructions.

5. The Table below gives the meanings for each of the control signals on the Y86 datapath.

Control	**Width**	**Description**
PCIncSrc	2	Determines value to add to PC to get to next instruction. 00 - 1, 01 - 2, 10 - 5, 11 - 6.
valCsrc	1	Determine value for valC. 0 means valC $\leftarrow$Dest (M_4[PC+1]), 1 means valC $\leftarrow$V/D (M_4[PC+2]).
valAsrc	1	Determine read register file output for valA. 0 means valA $\leftarrow$R[rA], 1 means valA $\leftarrow$R[%esp].
valBsrc	1	Determine read register file output for valB. 0 means valB $\leftarrow$R[rB], 1 means valB $\leftarrow$R[%esp].
dstEsrc	2	Determine destination register for write of valE. 00 means R[rB] $\leftarrow$valE, 01 means R[%esp] $\leftarrow$valE, 10 and 11 mean send 0xf to dstE, indicating no write.
dstMsrc	1	Determine destination register for write of valM. 0 means R[rA] $\leftarrow$valM, 1 means send 0xf to dstM, indicating no write.
aluAsrc	2	Determine value to send to ALU_A. 00 means ALU_A $\leftarrow$valA, 01 means ALU_A $\leftarrow$valC, 10 means ALU_A $\leftarrow$4, 11 means ALU_A $\leftarrow$-4.
aluBsrc	1	Determine value to send to ALU_B. 0 means ALU_B $\leftarrow$valB, 1 means ALU_B $\leftarrow$0.
setCC	1	Determine whether or not to update the CC register on the next clock. 0 indicates do not update, 1 indicates update.
aluOp	1	Determine operation to route to ALUfun. 0 means 0000 (add), 1 means use ifun.
dmemAddr	1	Determine address routed to data memory address line. 0 means dAddr $\leftarrow$valE. 1 means dAddr $\leftarrow$valA.
dmemData	1	Determine value routed to data memory D input. 0 means D $\leftarrow$valA, 1 means D $\leftarrow$valP
dmemWrite	1	Determine whether or not to store D at M_4[dAddr] on the next clock. 0 indicates do not write memory, 1 indicates write memory.
newPC	2	Determine source of next Program Counter to be routed to input of PC register. 00 means newPC $\leftarrow$valP, 01 means newPC $\leftarrow$valC, 10 means newPC $\leftarrow$valM, and 11 is undefined.

6. Steps to execute one or more instructions:

   (a) Begin at the Datapath circuit on the canvas with the pointer tool selected. Open `Y86.circ` and double-click the Datapath circuit in the explorer pane to make it the active circuit if this is not the case.

   (b) Right-click on the IMem device and select the 'View Memory' option.

   (c) For each of the memory banks, from top to bottom, you should right-click on the memory device and select the 'Load Image...' option.

   (d) Navigate in the file system and select the memory image appropriate to the particular bank (0, 1, 2, or 3).

   (e) Double-click on the Datapath circuit again to make it the active circuit.

   (f) Select the 'hand' tool so that we can manipulate values on the Datapath.

   (g) Take the clock to an asserted level.

   (h) Assert and then deassert the Reset pin. At this point, the PC, CC, and registers are initialized to zero and you should be able to see the values derivative of the first instruction at the probe displays (like icode, ifun, rA, and rB).

   (i) Instruction execution occurs in a single clock cycle. Once the instruction is available from IMem following the rising edge of the clock, we need to, based on the particular instruction, set the pins associated with the control signals in the datapath. Eventually, this manual operation will be replaced by a combinational circuit that performs the same task. For now, with the clock still high, set the control signals PCIncSrc, valCsrc, valAsrc, valBsrc, dstEsrc, dstMsrc, aluAsrc, aluBsrc, setCC, aluOp, dmemAddr, dmemData, dmemWrite, and newPC to the correct values based on the instruction being executed. The professor will take you through the examples of irmovl.

   (j) Once the control signals have been set, check the values from left to right through the datapath. Based on the register transfer semantics of the various instructions given in Tables 4.18 through 4.21 in your book, do the values of valP, valC, valA, valB, etc. correspond to the specific instruction being executed? If they do not, then verify your control signals again.

   (k) When the values/control signals are correct, change the clock to come down to the deasserted state. Note that when the clock goes low, the negation in the lower left of the datapath will cause the clock input to the register file, the CC register, and the data memory to go high. So by bringing the clock low, these devices will take any write-values on their input and store them into their device. This, in effect, "seals the deal" for the execution of this instruction and we are ready to go on to the next instruction.

   (l) Change the clock back to the high/asserted state. The value of newPC is now stored in PC and we go back to step 6i and set the control signals appropriately.

7. To complete this lab you must do the following:

    (a) You must write a Y86 assembly program with example instructions that cover the list of different instruction types given on the table (see last page). You may choose to write a single program with all instruction types or you may break it into a couple of smaller pieces. Make sure your name is in a comment block at the top of each program. Assemble each of these example programs using the `yas` tool we installed previously; that should produce a `.yo` file of the same name.

        You must then load the instructions for this program(s) into your instruction memory (and data memory if necessary). You can do this by creating the memory file images by hand or by using the `yo2banks.c` program to do it automatically. Here is an example appropriate for our system:

        ```
        > yo2banks -f rrmov.yo -b rrmov -c 4 -l 2
        ```

        This will produce `rrmov0.mem`, `rrmov1.mem`, `rrmov2.mem`, and `rrmov3.mem` which can be loaded into the memory modules.

    (b) You must execute at least one actual instruction of each type in the table on the last page. Make sure that the instruction executed correctly by verifying the intended results. Then copy your control signals in the table provided. You should fully specify all the signals using X for don't care where appropriate. An example instruction in the table is given for you. You must complete the others. Please follow the same format by putting the value on the top line of each cell and a mini-description on the bottom line of each cell. The top line value determines the correctness while the bottom line will help you to remember why that value is set that way.

        You might first complete this table with pencil. Please then download the `chart.tex` file from the class webpage and then input your answers there. Latex the file and submit to me the pdf of your table. Be sure to put your name in the table!

It is critical that you resist the temptation to take a short cut by only filling in what you believe to be the correct values. You really need to actually execute each instruction type to be sure that the values you put in for the control signals obtain the desired result. In previous years, students who have tried to fill in the control signals without testing them on actual instructions typically make many mistakes and earn a poor grade. If you are careful and thorough, it is easy to complete this lab without any mistakes to earn a perfect score.

This lab takes more time than you think. Be sure to budget your time to complete a couple of instructions per day. Aim to complete the lab two days prior to the deadline in order to leave yourself time to overcome unexpected difficulties. It is critical to complete this lab correctly because our next lab is to design the circuitry to automatically set these same control signals.

I would highly recommend following this schedule:

Easier	**Medium**	**Harder**
rrmovl	rmmovl	call
OP	mrmovl	ret
irmovl	jmp	pushl
nop	cmov	popl
halt		

Note that we don't have the "logic hardware" in this circuit to decide whether or not to do conditional moves and conditional jumps. Later we'll tie in the CC bits to help us determine what to do. For the purposes of this lab, assume the condition is true and set the control inputs to take the jump or do the move.

8. Table for you to complete

Instruction	PCIncSrc	valCsrc	valAsrc	valBsrc	dstEsrc	dstMsrc	aluAsrc	aluBsrc	setCC	aluOp	dmemAddr	dmemData	dmemWrite	newPC
OPl rA,rB														
rrmovl rA,rB														
irmovl $0x100,%esp irmovl V,rB	11 +6	1 V	X	0 rB	00 rB	1 F	01 valC	1 0	0 no	0 add	X	X	0 no	00 valP
rmmovl rA,D(rB)														
mrmovl D(rB),rA														
pushl rA														
popl rA														
jXX Dest														
cmovXX rA, rB														
call Dest														
ret														
nop														
halt														