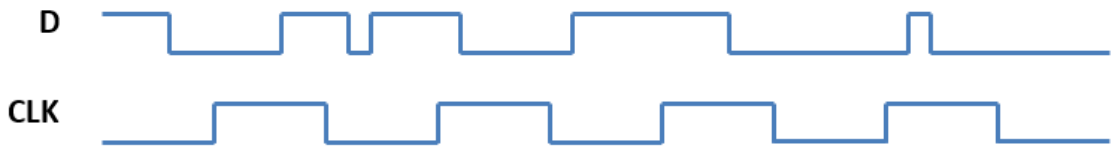
Jordan Wu

U0900517

Wednesday, March 11, 2015

Homework Assignment #7

1. The following diagram shows how two logical signals vary over time. Complete the diagram to show the Q output of a D latch and a D flip flop over time:



D Latch

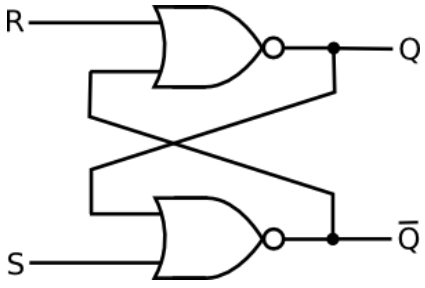
D Flip Flop

You should assume the D latch is transparent when the clock is high, and that the D flip flop latches data on a falling clock edge.

D Flip-flops and D Latch are the simplest memory elements, the outputs is equal to the value of the stored state inside the element. Both require a clock input to trigger the change of state. The only difference between them is when the state are change by the clock. For the D Latch the state is change when the clock is asserted. The D Flip-flops state are changed only at the clock edges. D Flip-flops and D Latch are used to store values in the internal memory

D Latch

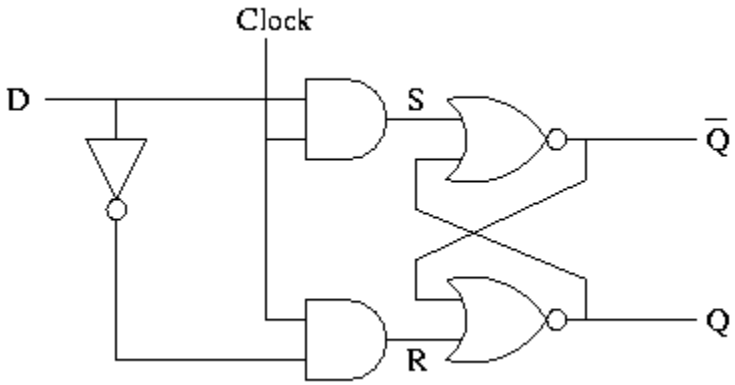
Here is a figure of an RS Latch.



Inputs		Outputs	
Reset	Set	Q	$\bar{Q}$
0	0	Q	$\bar{Q}$
0	1	1	0
1	0	0	1
1	1	X	X

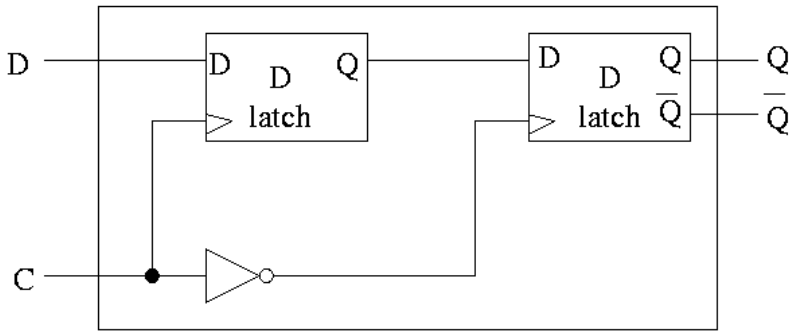
Here is a quick explanation on how the RS Latch works. If both Reset and Set are 0 then it hold the current state which is Q. If the Reset is 1 and the Set is 0, then it will reset Q to 0. If the Reset is 0 and Set is 1, then it will set the Q to 1. Finally when both Reset and Set are 1, it will be undefined which will not happen.

It is important to understand RS Latch since the D Latch uses this with two inputs, which are the data value and clock. D latches has two input and two outputs. The two inputs are the data value to be stored (D) and a clock signal (C). The clock again will determine when the latch should read the value on the D input and store it. The outputs are the internal state Q and its complement $\bar{Q}$. Here is a figure of the D-latch along with the RS Latch.



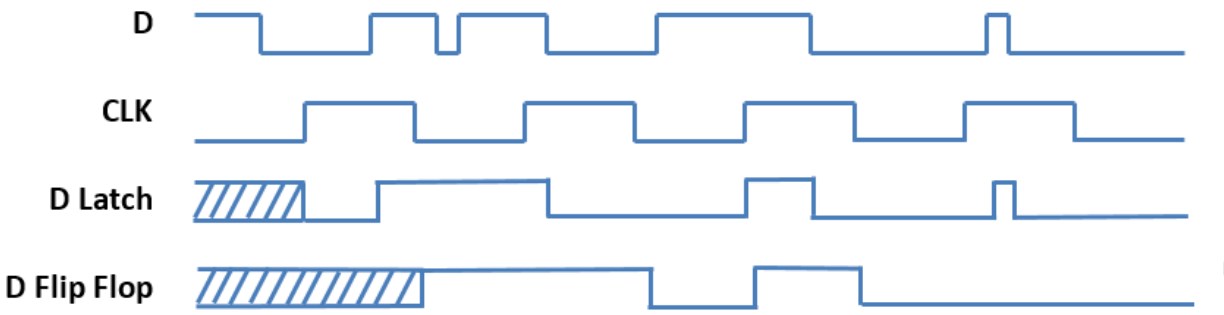
D Flip-Flops

The D Flip-flops are constructed from a pair of D Latches. Remember that the D Latch depends on the clock to store data (the inverting clock will store data when the clock is low). This allows the state Q to only be changed on the clock edge (falling edge). Here is a figure of a D Flip-flops.



Note: D flip-flops are built so that it can be trigger by either the rising or falling clock edges. We are assuming that the outputs are change only on the falling edges.

Here is the resulting timing diagram.

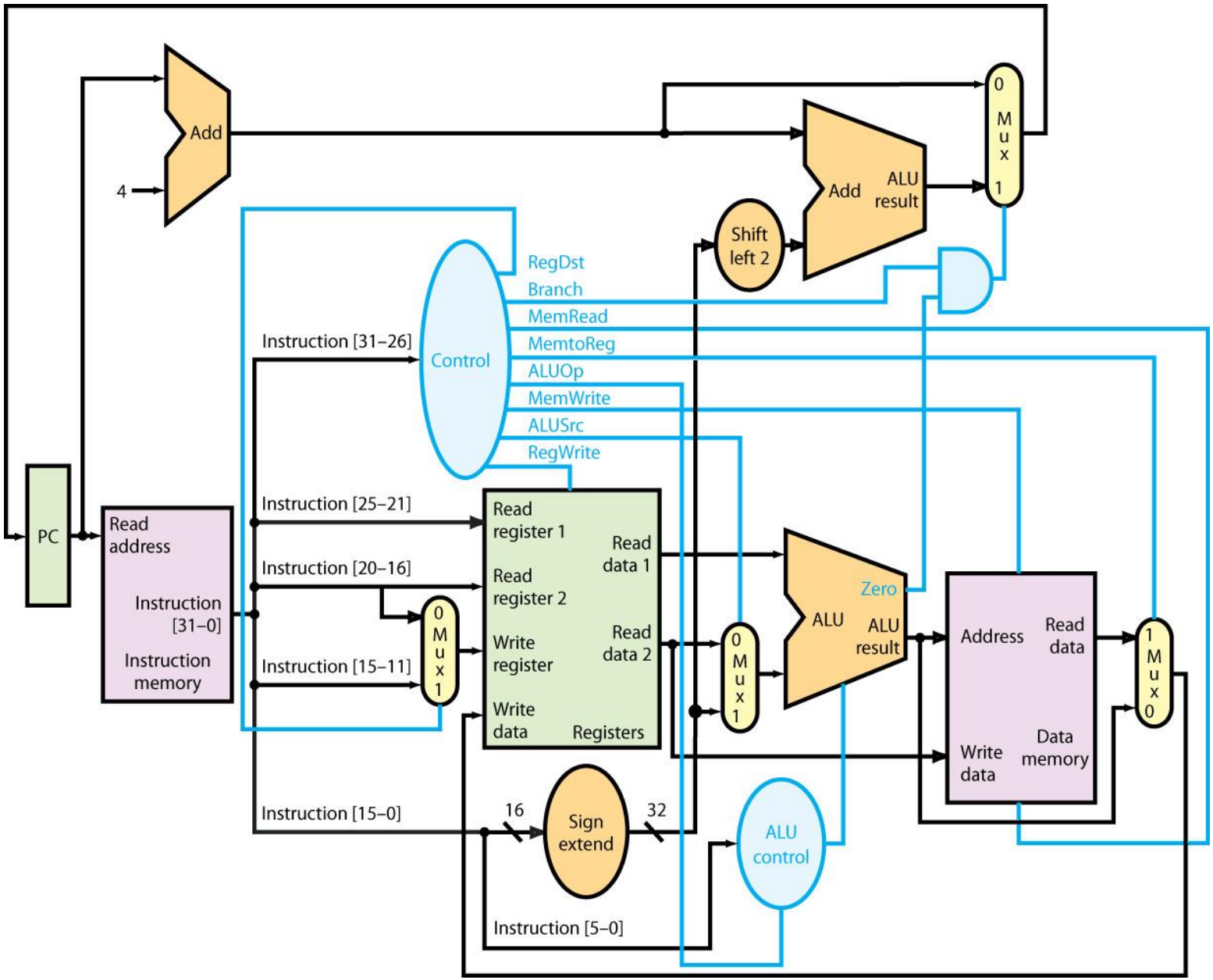


2. Different instructions use different parts of the single-cycle processor design. For each instruction type below, determine how that instruction must set the control lines. Fill in a small truth table that lists the instructions vertically and the control line labels horizontally. Write 0, 1, or x (don't care) for each instruction for each of the seven binary control lines from the CPU diagram. For the ALUOp control line, write down what computation the ALU should do *in English* (addition, etc.) for that instruction.

```
beq  Rs, Rt, offset
add  Rd, Rs, Rt
andi Rt, Rs, immediate
lw   Rt, offset(Rs)
sw   Rt, offset(Rs)
slt  Rd, Rs, Rt
```

Hint: Figure 4.18 shows a partial answer for some of these instructions.

Looking at the textbook we can see the following Control lines for the different instruction. We will be using the following CPU diagram below to determine the control lines for each of the instruction above. Note that the control lines in the diagram are determined by the opcode fields of the instruction. In the diagram everything that are the control unit are in blue and outputs seven control signals plus the ALUOp which controls the ALU results.


Add
4
Add
ALU result
0
M
u
x
1
Shift left 2
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
PC
Read address
Instruction [31–0]
Instruction memory
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
ALU
Zero
ALU result
Address
Read data
Write data
Data memory
16
Sign extend
32
ALU control

Each of the following control signal does the following:

- RegDst: The register destination number for the **Write register** comes from the rt field (20:16) when deasserted. Otherwise the register destination number for write register will be from rd field (15:11)
- RegWrite: When asserted the register on the **Write register** input is written with the value on the **Write data** input.
- ALUSrc: This control the second ALU operands, when deasserted this will use the **Read data 2** as the second operands, otherwise it will use the lower 16 bits (sign-extended).
- MemRead: When it is asserted the **Data memory** contents designated by the address input are put on the **Read data** output.
- MemWrite: When asserted the **Data memory** contents designated by the **Address** input are replaced by the value on the **Write data** input.
- MemtoReg: When deasserted the value fed to the register **Write data** input comes from the ALU, otherwise the value fed to the register **Write data** input comes from the **Data memory**.
- Branch: This is used for the branch-on-equal instruction which will branch if both the Branch control signal and **Zero** are 1.
- ALUOp: This control signal is used to control the ALU which will be determined by the funct field of the instruction.

Understanding each of these control signals will help determine which ones needs to be asserted or deasserted for each instructions. I will quickly go through each of the instruction stated above and determine which control signals are asserted and deasserted.

beq:

This is will be an I-type instruction which will need to compare the rs with rt for equality. To check for equality the ALUOp must use subtraction. If the value in both register are equal then the Zero output from the ALU will be asserted. If it is asserted then a branch is taken by adding the PC = PC + 4 + BranchAdr. The BranchAdr is the offset which is the lower 16 bits 15:0 sign extended and need to be shifted 2 bits to the left (factor of 4) before adding to PC + 4. Don't care about the RegDst or the MemtoReg since this instruction does not involve writing data. The other controls signal will be deasserted.

add:

Add instruction is an R-type instruction which will need to use all three of the registers. Since the Write register is rd, the RegDst must be asserted and the ALUOp is addition. There is no need to read or write from the Memory so the MemtoReg much be deasserted which allow the ALU result as the Write data to store in the register destination rd.

andi:

Andi instruction is an I-type instruction which will need to use the lower 16 bit of the instruction as the immediate value, ALUSrc must be asserted. The register destination for this instruction is rt, this mean that the RegDst must be deasserted. The ALUOp must be AND for this instruction to function correctly. Lastly the ALU result must be write into the register destination rt, to do this the MemtoReg must be deasserted.

lw:

Load word is an I-type instruction that will need to read data from Memory. This instruction has the register destination as rt which mean the RegDst must be deasserted. The immediate value is the lower 16-bit of the instruction that much be added to the base address which is stored in the rs register, this mean ALUOp is add and ALUSrc is asserted. Both the MemRead and MemWrite must be asserted.

sw:

Store word is an I-type instruction that will need to write data into Memory. This instruction used this lower 16 bits of the instruction and will need to use addition in the ALU, ALUOp is add and ALUSrc is asserted. MemWrite must be asserted to allow the ALU results which is the address to be store into Memory.

slt:

Set less than is an R-type instruction that is used to compare two value. This instruction uses three register which the register destination is rd, RegDst must be asserted. The ALUOp will be set on less than and the MemtoReg is deasserted to be able to write data into the register. RegWrite is asserted.

Here is the truth table for the control line for the list of instructions

Instruction	Control Lines							
	RegDst	Branch	MemRead	MemtoReg	ALUOp	MemWrite	ALUSrc	RegWrite
beq	X	1	0	X	subtract	0	0	0
add	1	0	0	0	add	0	0	1
andi	0	0	0	0	AND	0	1	1
lw	0	0	1	1	add	0	1	1
sw	X	0	0	X	add	1	1	0
slt	1	0	0	0	set on less than	0	0	1

3. Using the same CPU diagram, assume that the logic blocks in the processor have the following latencies and constraints:
   - The PC register will be ready to output data at the beginning of the clock cycle.
   - The control logic latency is 30 ps; control logic outputs will be correct 30 ps after all inputs arrive.
   - MUX latency is 20 ps; all MUXs outputs will be correct 20 ps after the last relevant input arrives.
   - Any ALU that only does addition has a latency of 70 ps.
   - The general purpose ALU has a latency of 90 ps.
   - Instruction memory has a latency of 200 ps, and data memory has a latency of 250 ps for reads or writes.
   - The register file has a read latency of 80 ps.
   - Any data that is to be written back to any register (register file / PC) must arrive at the register file 10 ps before the end of the clock cycle. (This is called 'setup time', and it allows the first stage latch of the flip-flops enough time to capture the data before the edge of the clock arrives.)
   - Sign extension and shift left 2 have insignificant latency. (In reality, they'd have a little due to wire delays and the need for signal amplification, but we'll ignore it.)

beq: SHORTEST

PC register → Instruction memory → Register file → MUX latency → MUX output → ALU → MUX output → Setup Time

0 → 200 → 280 → 300 → 320 → 410 → 430 → 440 ps

add:

PC register → Instruction memory → Control → RegDst → MUX output → Register file → MUX latency → MUX output → ALU → MUX latency → MUX output → Setup Time

0 → 200 → 230 → 260 → 280 → 360 → 380 → 400 → 470 → 490 → 510 → 520 ps

andi:

PC register → Instruction memory → Control → RegDst → MUX output → Register file → MUX latency → MUX output → ALU → MUX latency → MUX output → Setup Time

0 → 200 → 230 → 260 → 280 → 360 → 380 → 400 → 490 → 510 → 530 → 540 ps

lw: LONGEST

PC register → Instruction memory → Control → RegDst → MUX output → Register file → MUX latency → MUX output → ALU → Data memory → MUX latency → MUX output → Setup Time

0 → 200 → 230 → 260 → 280 → 360 → 380 → 400 → 470 → 720 → 740 → 760 → 770 ps

sw:

PC register → Instruction memory → Register file → MUX latency → MUX output → ALU → Data memory → MUX latency → MUX output

0 → 200 → 280 → 300 → 320 → 390 → 640 → 660 → 680 ps

slt:

PC register → Instruction memory → Control → RegDst → MUX output → Register file → MUX latency → MUX output → ALU → MUX latency → MUX output → Setup Time

0 → 200 → 230 → 260 → 280 → 360 → 380 → 400 → 490 → 510 → 530 → 540 ps

Consider only the instructions that our processor supports. Which instruction has the longest latency? Show your work.

Looking at the previous page the **load word** instruction seem to take the longest. This is because it needs to access the data memory along with a setup time to write back into the destination register. This means that this instruction goes there more component than any other instruction. The work is on the previous page.

Which instruction has the shortest latency? Show your work.

Looking at the previous page the **branch on equal** instruction has the shortest latency. The reason for this is that it does not access the data memory and also doesn't not need to write into a register. The only thing it needs to do is check for equality by subtracting the two data read from the rs and rt and uses that as a selector for selecting the correct instruction address.

What determines the clock cycle for a CPU, and what frequency would our CPU design run at? Show your work.

We know that the longest latency is the lw instruction which needs 770 ps which is the clock cycle which can be converted into frequency.

$$clock\ rate = \frac{1}{clock\ cycle} = \frac{1}{770 \cdot 10^{-12}} = 1298701299\ Hz \approx 1.3\ GHz$$

4. Using a copy of the CPU diagram, modify it so that the single-cycle CPU can execute the jal address instruction and the jr Rs instruction. Justify your changes in a few sentences:

First of all we are now dealing with J-type instruction which have the following format.

opcode	address
6 bits	26 bits

Implementing J-type instruction involves changing the PC somewhat like the branch instruction except that it is not conditional. The low-order 2 bits of jump address are always 00_{two} (the reason is that the offset must be a factor of 4 so shifting the bits left by two). The address is 26 bits will be added to the left of 00_{two}. The remaining 4 bits will come from the upper 4 bits of the PC + 4. Concatenation of these will implement the jump instruction.

- The upper 4 bits of the current PC + 4
- The 26-bit immediate field of the jump instruction
- The bits 00_{two}

Current PC + 4	Immediate field of jump instruction	shift to the left by 2
4 bits	26 bits	2 bits

What logic blocks did you have to add or modify?

- An **additional multiplexor** that is used to select the source for the new PC value. This multiplexor will determine whether it should **(branch/PC+4)** (the multiplexor before this determines if branches or not) **or jump.** (Control signal called Jump)

- An **additional multiplexor** that is used to select the source for the new PC value. This multiplexor will determine whether it should **jump to address or jump to register**. This multiplexor will be after the additional multiplexor above. (Control signal called JumpReg)
- An **additional multiplexor for the Write data** to allow a selector for which data to write, **(ALU result/data memory) or PC + 4**. (Control signal called WritePC)
- **Modify the existing multiplexor that controls the Write register**, will need **one addition input** which is the value for **$ra ($11111_{two}$)**.

Why did you add additional data lines?

- An **additional data line** for the PC must be modify so that the instruction bits can be concatenated with the PC + 4 bits, **from the instruction to a shift left by two concatenated with the first 4 bits of the PC+4 into the multiplexor** that determines if it jumps or not.
- An **additional data line** that will allow to jump to a register, **from the Read data 1 output of the register file to the multiplexor** that controls whether or not to jump to address or register.
- An **additional data line** to save the PC+4 into a register file, **from the PC+4 to a multiplexor** that determine if the Write data is PC+4 or (ALU result / Read data).
- An **additional data line** to allow the PC+4 to be saved into $ra, **include a data line $ra into the multiplexor** that control the Write register.

Did you have to add any control signals?

- **Jump**: This signal will determine whether or not the instruction is a J-type instruction. When asserted a jump has been taken, otherwise there was not jump.
- **JumpReg**: This signal will determine whether to jump to address or to register. When asserted the new PC will be the register Rs, otherwise it would be the jump address.
- **WritePC**: This signal will determine whether to write data from the ALU result/Read data or write the PC + 4. When asserted this will write the PC+4 into the destination register, otherwise the ALU result/Read data will be written into the destination register.
- **RegDst**: This signal will be modify since it will now have 3 inputs. This signal is 2 bits. The following table below will determine how this control signal selects the three registers.

Registers	RegDst
rt	00
rd	01
ra	10

control signals for the following four instructions

Instruction	Control Lines										
	RegDst	Jump	JumpReg	WritePC	Branch	MemRead	MemReg	ALUOp	MemWrite	ALUSrc	RegWrite
jal	10	1	0	1	0	0	X	X	0	X	1
j	X	1	0	X	0	0	X	X	0	X	0
jr	X	1	1	X	0	0	X	X	0	X	0
jalr	10	1	1	1	0	0	X	X	0	X	1