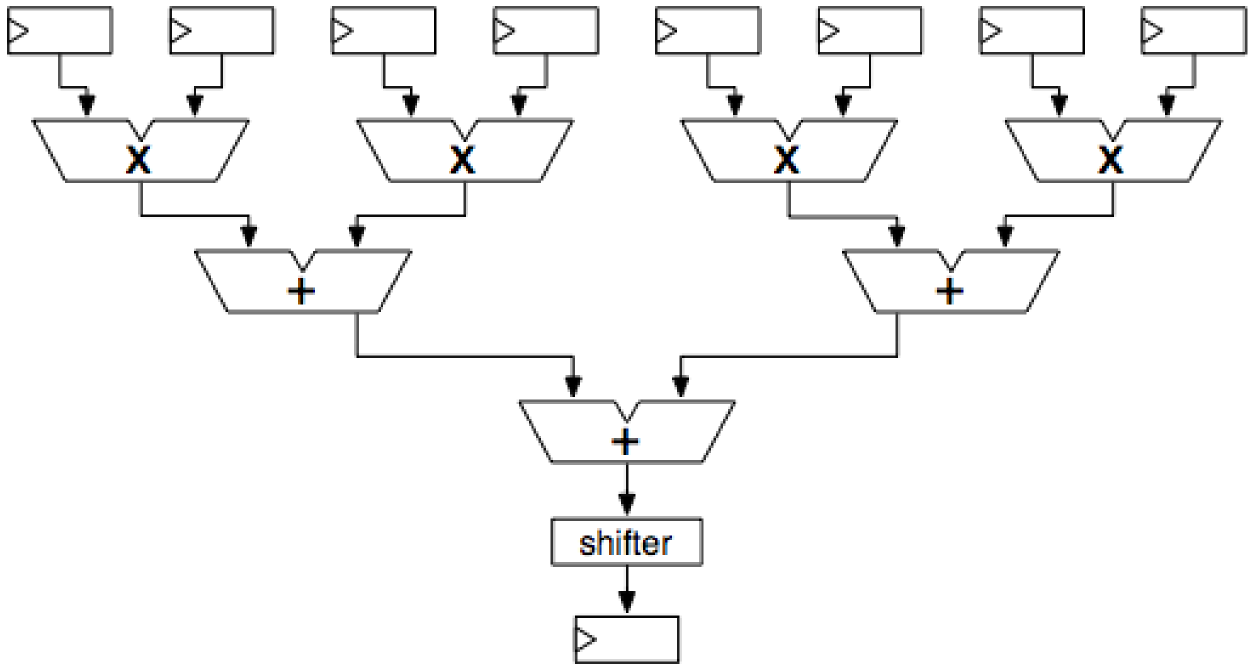
CS61C Fall 2013 – 10 – Clocking and CPU Design

Clocking Methodology

- The input signal to each state element must stabilize before each rising edge.
- Critical path: Longest delay path between state elements in the circuit.
- Min clock period = $t_{clk\text{-}to\text{-}q} + t_{CL} + t_{setup}$, where t_{CL} is the Combinational Logic delay in the critical path.
- If we place registers in the critical path, we can shorten the period by reducing the amount of logic between registers

Clocking Problem

- The circuit below computes the weighted average of 4 values
- Logic Delays - t_{mult} = 55ns, t_{add} = 19ns, t_{shift} = 2ns
- Register Parameters - t_{setup} = 2ns, t_{hold} = 1ns, $t_{clk\text{-}to\text{-}q}$ = 3ns



1. What is the critical path delay and the maximum clock rate this circuit can operate at?

Critical path – path from a top register to the bottom register.
Clk-to-q + mult + add + add + shift + setup = 3 + 55 + 19 + 19 + 2 + 2 = 100 ns.
Max frequency = 1/Min period = 10 MHz

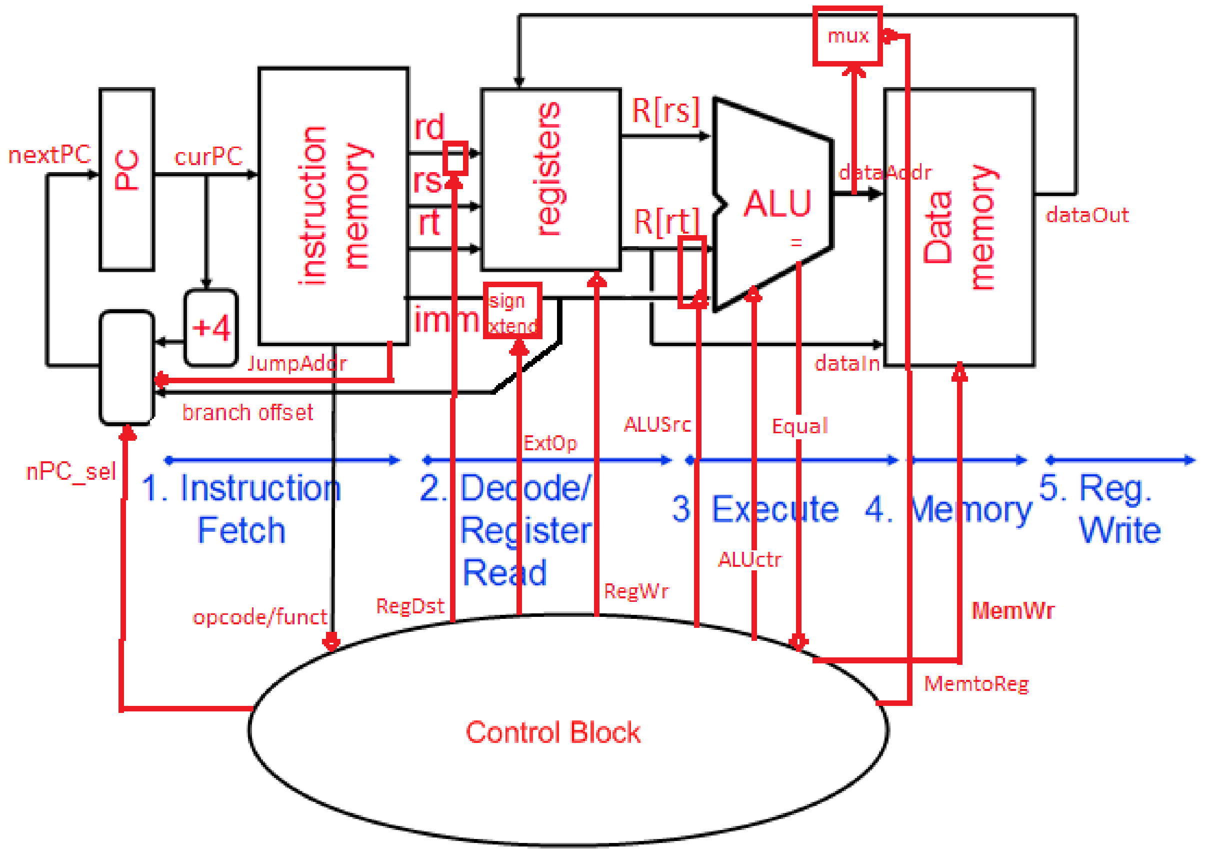
2. If you add one stage of registers (pipelining), what is the highest clock rate you can get?

Best to add registers in a place that minimizes the longest path through the circuit. This would be right after the multiplication.
Path from top registers to middle registers is clk-to-q + mult + setup = 3 + 55 + 2 = 60 ns.
Path from middle to bottom registers is clk-to-q + add + add + shift + setup = 3 + 19 + 19 + 2 + 2 = 45 ns.
So our new critical path has Min period of 60 ns, so our Max frequency = 16.67 MHz.

CPU Design

`rd`, `rs`, and `rt` are 5-bit wires, `imm` is a 16-bit wire. All other wires are 32 bits wide. Assume that the ALU can output an Equals signal, which is on when its two inputs are equal.

1. Add control signals and missing elements (such as multiplexers) to the diagram below so that the datapath can execute the following instructions: `add, lui, sw, bne, j.`



j: Need to get the 26-bit target address from the instruction, hence we add a new line **`JumpAddr`** from Instr Mem to the "address logic" module that computes the `nextPC` (rounded rectangle in lower-left). In addition, we need a way for the address logic module to choose between a step, a branch, and a jump, so we add the control signal **`nPC_sel`** to MUX them. A jump instruction only updates the PC, so we actually don't want to write to the Register File or Data Mem. We add control signals **`RegWr`** and **`MemWr`** to allow us to designate when we want to write or not.

add: Two registers are read and added, then the result is stored back into another register. The Register File can already output the values of the correct two registers (`R[rs]` and `R[rt]`) and the ALU can do addition, so now we need to make sure we can store the result back into a register. First we add the control signal **`ALUctr`** to select the proper ALU operation. Currently, the register input is `dataOut`, but the result we want is the output of the ALU. So we add a MUX on the register input line that selects between `dataOut` and `ALUOut` using the new control signal **`MemtoReg`**.

lui: Shift the `imm` left by 16 bits, then store into register. The shifting can be done by the ALU, but it needs the `imm` instead of `R[rt]`, so we MUX those signals together and use the control signal **`ALUSrc`** to pick between them. The result of the ALU is stored back into a register. Unlike `add`, though, I-format instructions store into `$rt` instead of `$rd`. So we add a MUX for the `rd` input to the Reg File that selects between `rd` and `rt` based on the new control signal **`RegDst`**.

sw: Store `R[rt]` into Data Mem at address `R[rs] + SignExtImm`. `R[rt]` is already set up as `dataIn` to Data Mem, so we just need to sign extend the immediate and let the ALU do the addition. We add **a sign extend block** on the `imm` line (the **`ExtOp`** signal is only necessary if we also have an instruction that requires ZeroExtImm, such as ori, but is not necessary here).

bne: Branch if `R[rs] != R[rt]`. We assume the proper `nextPC` is calculated from the branch offset in the address logic module. The ALU performs the comparison of `R[rs]` and `R[rt]` and we use the signal **`Equal`** as an *input* to the control logic, which will help determine the value of `nPC_sel`.

2. Fill out the values for the control signals from part 2 (write the names of your control signals in the second row):

Instr	Control Signals						
	nPC_sel	**RegDst**	**RegWr**	**ALUSrc**	**ALUctr**	**MemtoReg**	**MemWr**	**ExtOp**
add	PC+4	rd	1	rt	add	0	0	X
lui	PC+4	rt	1	imm	shiftx16	0	0	Zero*
sw	PC+4	X	0	imm	add	X	1	Sign
bne	branch	X	0	rt	X	X	0	Sign**
j	jump	X	0	X	X	X	0	X

* Doesn't really matter since `ALUctr` is shiftx16, so could set to `Sign` and eliminate `ExtOp`
** Branch also multiplies immediate by 4 at some point

3. Suppose you wanted to add a new instruction, `beqr`, which will be used like this: `beqr $x, $y, $z` will branch to the address in `$z` if `$x` and `$y` are equal, otherwise continue to the next instruction. Show any changes that would need to be made to the datapath above to make this instruction work.

The diagram is already crowded, so I will just explain the changes. Since we're reading three registers at once here, we need to add a third read port to the register file, and correspondingly, a third read address port. We can reuse `beq`'s datapath to compare two registers, `rs` and `rt`, so `rd` would contain the address to jump to. We would then connect `R[rd]` to the address logic module.