EECS 370 - Lecture 11

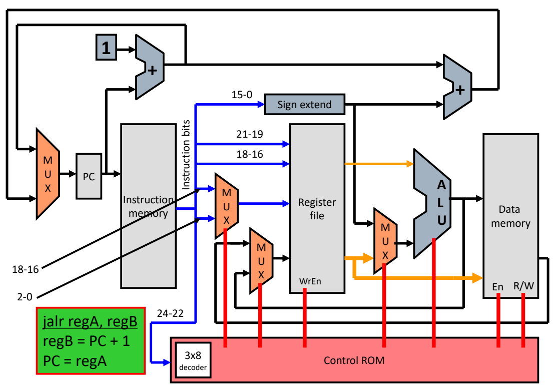
Multi-Cycle Data Path

Live Poll + Q&A: slido.com #eecs370 *Poll and Q&A Link*

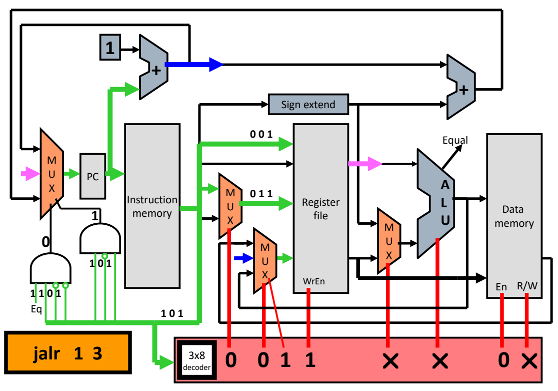
So Far, So Good

- Every architecture seems to have at least one "ugly" instruction
  - Something that doesn't elegantly fit in with the hardware we've already included
- For LC2K, that ugly instruction is JALR
  - It doesn't fine into our nice clean datapath
- To implement JALR we need to:
  - Write PC+1 into regB
  - Move regA into PC
- Right now there is:
  - No path to write PC+1 into a register
  - No path to write a register to the PC

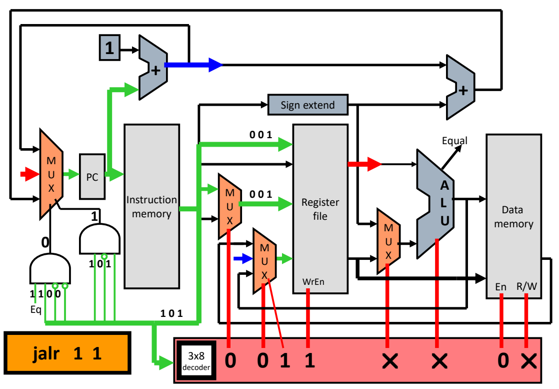
Executing a JALR Instruction



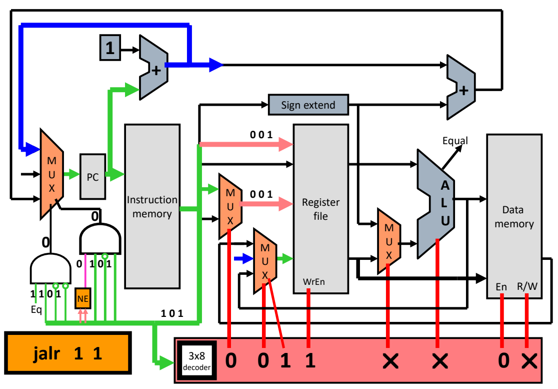
Executing a JALR Instruction



What if regA = regB for JALR?



Changes for JALR 1 1 Instruction



What's Wrong with Single-Cycle?

- **All instructions run at the speed of the slowest instruction.**
- Adding a long instruction can hurt performance
  - What if you wanted to include multiply?
- You cannot reuse any parts of the processor
  - We have 3 different adders to calculate PC+1, PC+1+offset and the ALU
- No benefit in making the common case fast
  - Since every instruction runs at the slowest instruction speed
    - This is particularly important for loads as we will see later

What's Wrong with Single-Cycle?

- 1 ns – Register read/write time
- 2 ns – ALU/adder
- 2 ns – memory access
- 0 ns – MUX, PC access, sign extend, ROM

Poll: What is the latency of lw?

	Get Instr	read reg	ALU oper.	mem	write reg	
• add:	2ns	+ 1ns	+ 2ns		+ 1 ns	= 6 ns
• beq:	2ns	+ 1ns	+ 2ns			= 5 ns
• sw:	2ns	+ 1ns	+ 2ns	+ 2ns		= 7 ns
• lw:	2ns	+ 1ns	+ 2ns	+ 2ns	+ 1ns	= 8 ns

Computing Execution Time

Assume: 100 instructions executed
- 25% of instructions are loads,
- 10% of instructions are stores,
- 45% of instructions are adds, and
- 20% of instructions are branches.

Single-cycle execution:

100 * 8ns = **800** ns

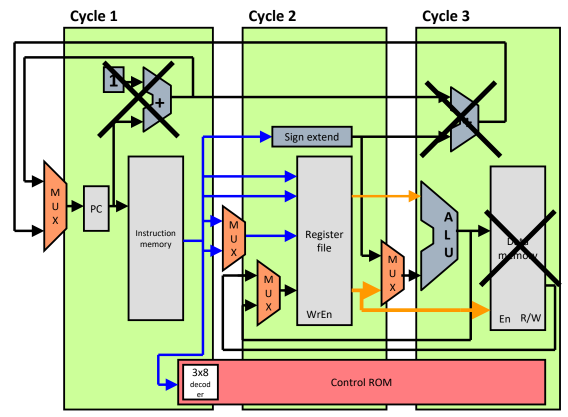
Optimal execution:

25*8ns + 10*7ns + 45*6ns + 20*5ns = **640** ns

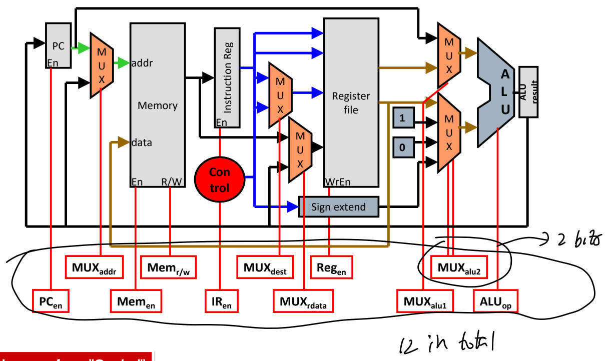
Multiple-Cycle Execution

- Each instruction takes multiple cycles to execute
  - Cycle time is reduced
  - Slower instructions take more cycles
  - Faster instruction take fewer cycles
    - We can start next instruction earlier, rather than just waiting
  - Can reuse datapath elements each cycle
- What is needed to make this work?
  - Since you are re-using elements for different purposes, you need more and/or wider MUXes.
  - You may need extra registers if you need to remember an output for 1 or more cycles.
  - Control is more complicated since you need to send new signals on each cycle.

LC2K Datapath – cycle groups

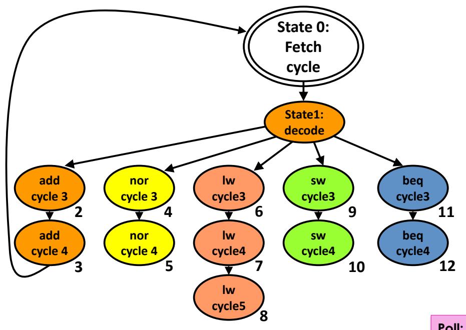


Multi-cycle LC2 Datapath



Each red signal comes from "Control" (implemented via ROM as before)

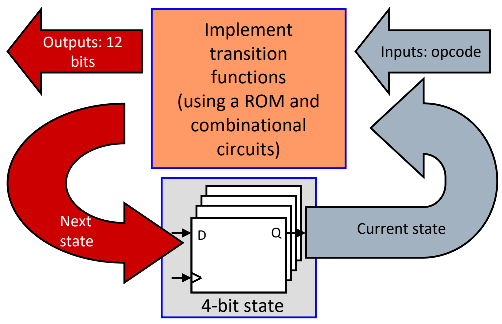
State machine for multi-cycle control signals (transition functions)



Note: we aren't worrying about JALR instruction in hardware going forward

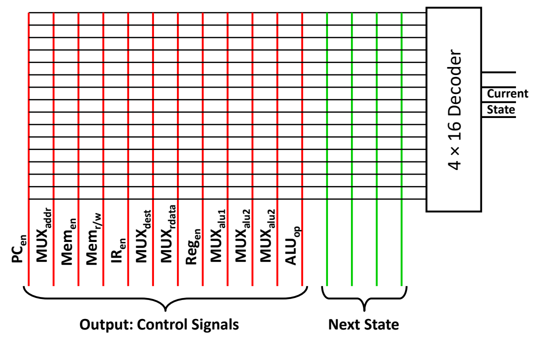
Poll: How many bits of storage are needed to store the state?

Implementing FSM



Outputs: 12 bits
Implement transition functions (using a ROM and combinational circuits)
Inputs: opcode
Next state
D
Q
Current state
4-bit state

Building the Control ROM

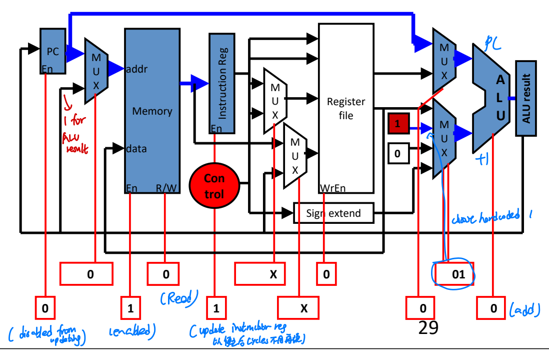


First Cycle (State 0) Fetch Instr

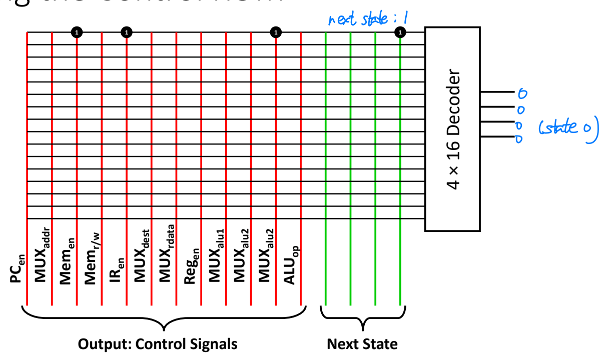
- What operations need to be done in the first cycle of executing any instruction?
  - Read memory[PC] and store into instruction register.
    - Must select PC in memory address MUX (MUX_{addr}= 0)
    - Enable memory operation (Mem_{en}= 1)
    - R/W should be (read) ($Mem_{r/w}$= 0)
    - Enable Instruction Register write (IR_{en}= 1)
  - Calculate PC + 1
    - Send PC to ALU (MUX_{alu1} = 0)
    - Send 1 to ALU (MUX_{alu2} = 01)
    - Select ALU add operation (ALU_{op} = 0)
- PC_{en} = 0; Reg_{en} = 0; MUX_{dest} and MUX_{rdata}= X
- Next State: Decode Instruction

First Cycle (State 0) Fetch Instr

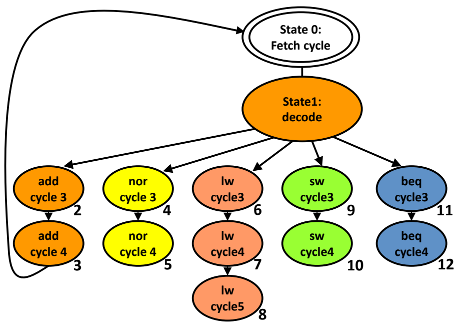
This is the same for all instructions
(since we don't know the instruction yet!)



Building the Control ROM

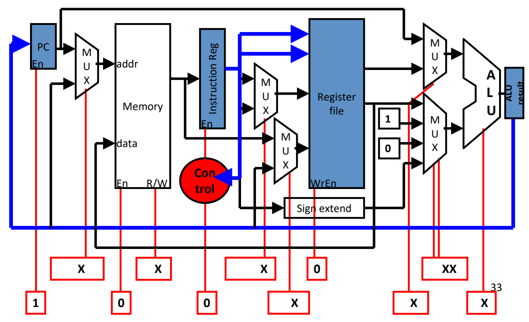


State 1: instruction decode



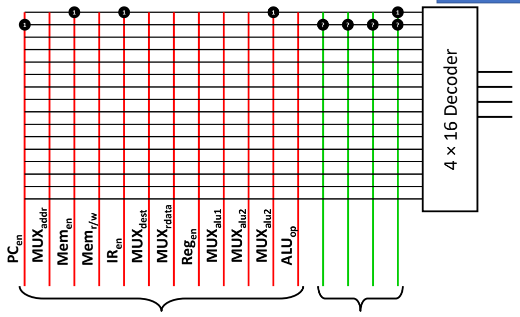
State 1: output function

Update PC; read registers (regA and regB);
use opcode to determine next state

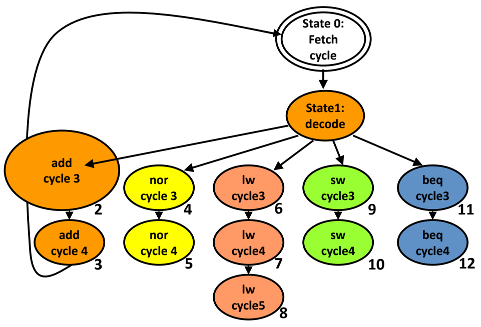


Building the Control ROM

How do we set next state?? Let's come back to that…

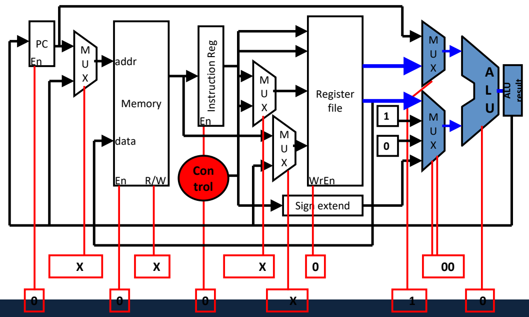


State 2: Add cycle 3

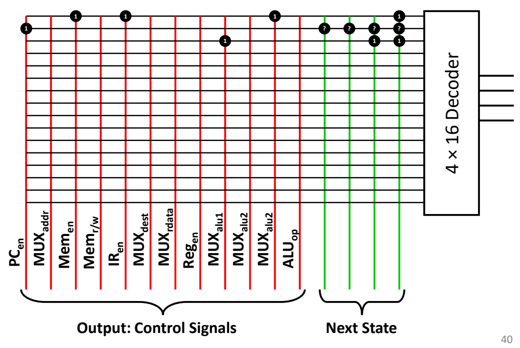


State 2: Add Cycle 3 Operation

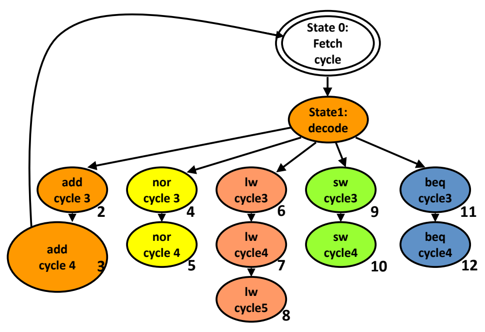
Send control signals to MUX to select values of
regA and regB and control signal to ALU to add



Building the Control Rom

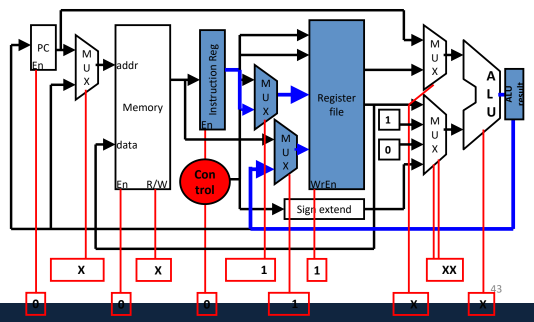


State 3: Add cycle 4

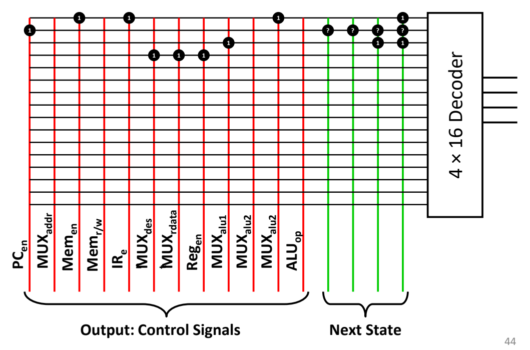


Add Cycle 4 (State 3) Operation

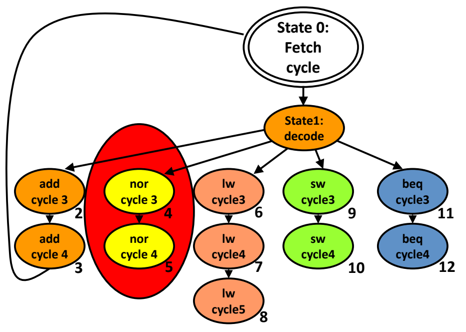
Send control signal to address MUX to select dest and to data MUX to select ALU output, then send write enable to register file.



Building the Control Rom

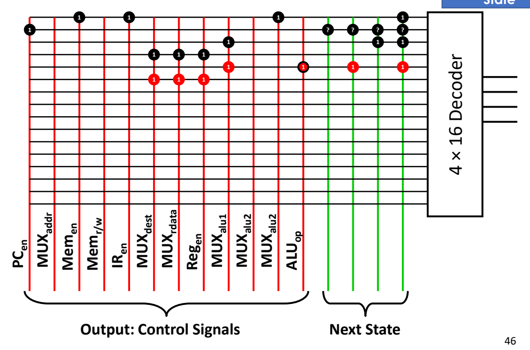


Return to State 0: Fetch cycle to execute the next instruction

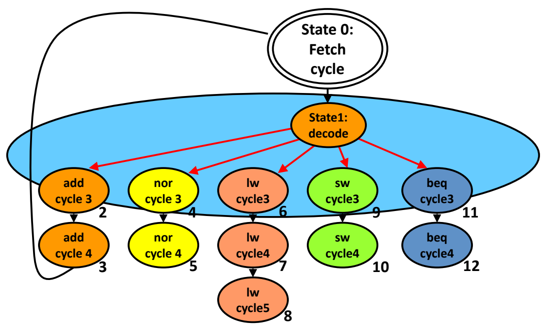


Control Rom for nor (4 and 5)

Same output as add except ALU_{op} and Next State

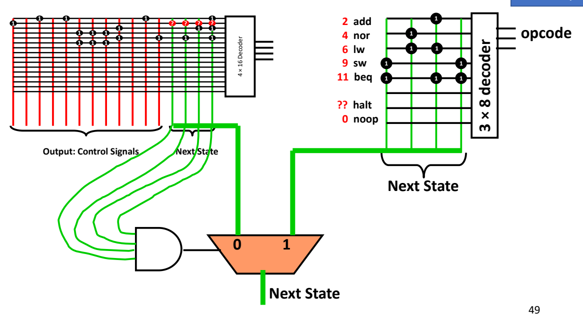


What about the transition from state 1?

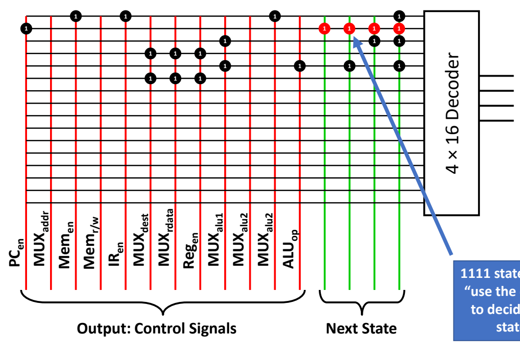


Complete transition function circuit

Secondary ROM stores which state we should branch to after decode for each opcode



Control Rom (use of 1111 state)



1111 state means "use the opcode to decide next state"