ECE/CS 552: Data Path and Control

Instructor:Mikko H Lipasti

Fall 2010
University of Wisconsin-Madison

Lecture notes based on set created by Mark Hill.

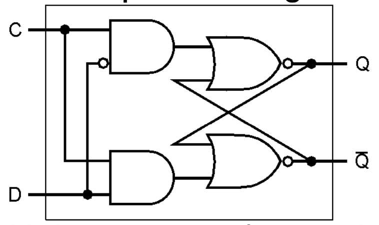
Processor Implementation

- Forecast – heart of 552 – key to project
  - Sequential logic design review (brief)
  - Clock methodology (FSD)
  - Datapath – 1 CPI
    - Single instruction, 2's complement, unsigned
  - Control
  - Multiple cycle implementation (information only)
  - Microprogramming
  - Exceptions

Review Sequential Logic

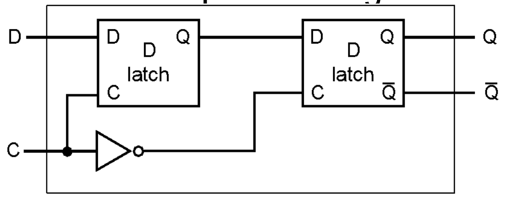
- Logic is combinational if output is solely function of inputs
  - E.g. ALU of previous lecture
- Logic is sequential or "has state" if output function of:
  - Past and current inputs
  - Past inputs remembered in "state"
  - Of course, no magic

Review Sequential Logic



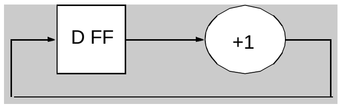
- Clock high, Q = D, ~Q = ~D after prop. Delay
- Clock low Q, ~Q remain unchanged
  - Level-sensitive latch

Review Sequential Logic



- E.g. Master/Slave D flip-flop
  - While clock high, Q_M follows D, but Q_S holds
  - At falling edge Q_M propagates to Q_S

Review Sequential Logic

- Can build: 

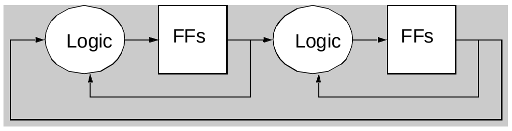
- Why can this fail for a latch?

Clocking Methology

- Motivation
  - Design data and control without considering clock
  - Use Fully Synchronous Design (FSD)
    - Just a convention to simplify design process
    - Restricts design freedom
    - Eliminates complexity, can guarantee timing correctness
    - Not really feasible in real designs
    - Even in 554 you will violate FSD

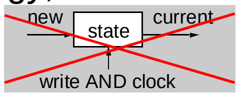
Our Methodology

- Only flip-flops
- All on the same edge (e.g. falling)
- All with same clock
  - No need to draw clock signals
- All logic finishes in one cycle

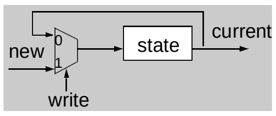


Our Methodology, cont'd

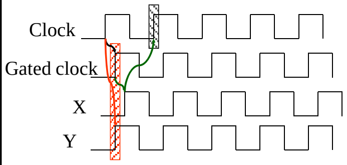
- No clock gating!
  - Book has bad examples

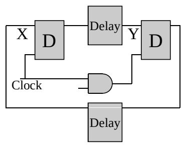


- Correct design:



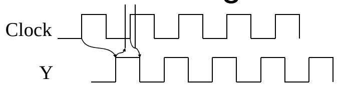
Delayed Clocks (Gating)

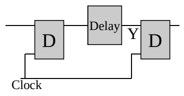




- Problem:
  - Some flip-flops receive gated clock late
  - Data signal may violate setup & hold req't

FSD Clocking Rules





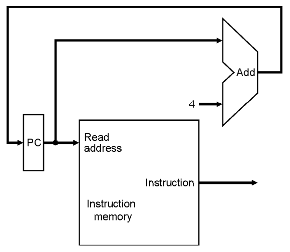
- T_{clock} = cycle time
- T_{setup} = FF setup time requirement
- T_{hold} = FF hold time requirement
- T_{FF} = FF combinational delay
- T_{comb} = Combinational delay
- FSD Rules:
  - $T_{clock} > T_{FF} + T_{combmax} + T_{setup}$
  - $T_{FF} + T_{combmin} > T_{hold}$

Datapath – 1 CPI

- Assumption: get whole instruction done in one long cycle
- Instructions:
  - add, sub, and, or slt, lw, sw, & beq
- To do
  - For each instruction type
  - Putting it all together

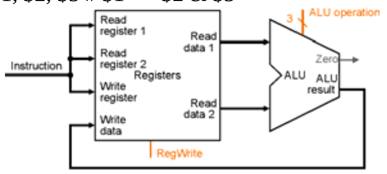
Fetch Instructions

- Fetch instruction, then increment PC
  - Same for all types
- Assumes
  - PC updated every cycle
  - No branches or jumps
- After this instruction fetch next one



ALU Instructions

- and \$1, \$2, \$3 # \$1 <= \$2 & \$3



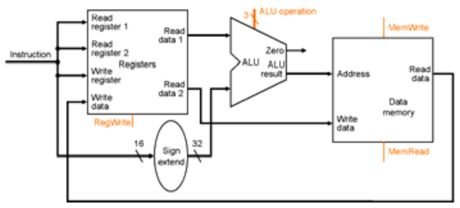
- E.g. MIPS R-format

Opcode	rs	rt	rd	shamt	function
6	5	5	5	5	6

Load/Store Instructions

- lw \$1, immed(\$2) # \$1 <= M[SE(immed)+\$2]
- E.g. MIPS I-format:

Opcode	rt	rt	immed
6	5	5	16

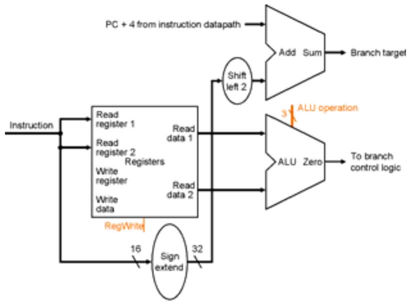


Branch Instructions

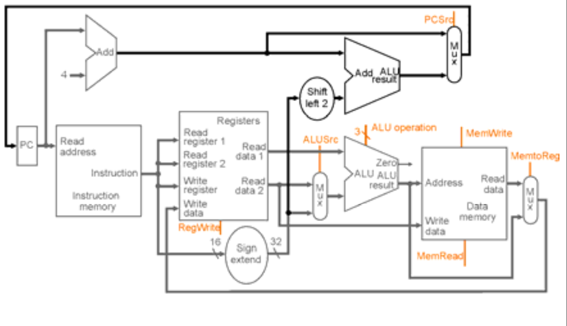
- beq \$1, \$2, addr # if (\$1==\$2) PC = PC + addr<<2
- Actually

```
newPC = PC + 4
target = newPC + addr << 2 # in MIPS offset from newPC
if ((\$1 - \$2) == 0)
    PC = target
else
    PC = newPC
```

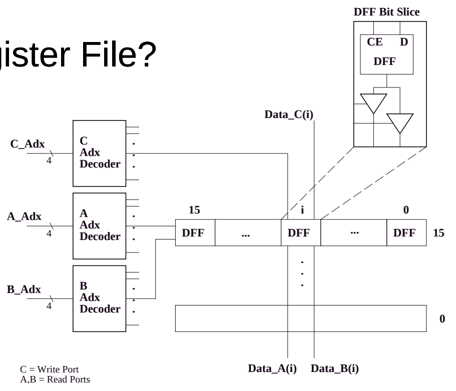
Branch Instructions



All Together



Register File?



Control Overview

- Single-cycle implementation
  - Datapath: combinational logic, I-mem, regs, D-mem, PC
    - Last three written at end of cycle
  - Need control – just combinational logic!
  - Inputs:
    - Instruction (I-mem out)
    - Zero (for beq)
  - Outputs:
    - Control lines for muxes
    - ALUop
    - Write-enables

Control Overview

- Fast control
  - Divide up work on “need to know” basis
  - Logic with fewer inputs is faster
- E.g.
  - Global control need not know which ALUop

ALU Control

- Assume ALU uses

000	and
001	or
010	add
110	sub
111	slt (set less than)
others	don’t care

ALU Control

Instruction	Operation	Opcode	Function
add	add	000000	100000
sub	sub	000000	100010
and	and	000000	100100
or	or	000000	100101
slt	slt	000000	101010

- ALU-ctrl = f(opcode,function)

But…don’t forget

Instruction	Operation	Opcode	function
lw	add	100011	xxxxxx
sw	add	101011	xxxxxx
beq	sub	000100	100010

- To simplify ALU-ctrl
  - ALUop = f(opcode)

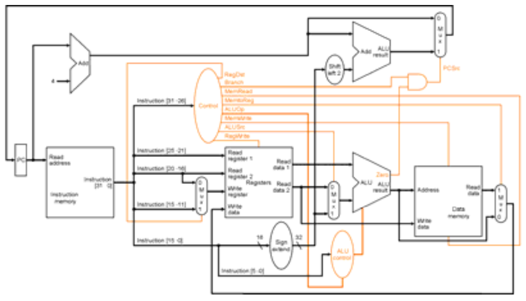
    2 bits 6 bits

ALU Control

10	add, sub, and, …
00	lw, sw
01	beq

- ALU-ctrl = f(ALUop, function)
- 3 bits 2 bits 6 bits
- Requires only five gates plus inverters

Control Signals Needed



Global Control

- R-format: opcode 6 | rs 5 | rt 5 | rd 5 | shamt 5 | function 6
- I-format: opcode 6 | rs 5 | rt 5 | address/immediate 16
- J-format: opcode 6 | address 26

Global Control

- Route instruction[25:21] as read reg1 spec
- Route instruction[20:16] are read reg2 spec
- Route instruction[20:16] (load) and and instruction[15:11] (others) to
  - Write reg mux
- Call instruction[31:26] op[5:0]

Global Control

- Global control outputs
  - ALU-ctrl - see above
  - ALU src - R-format, beq vs. ld/st
  - MemRead - lw
  - MemWrite - sw
  - MemtoReg - lw
  - RegDst - lw dst in bits 20:16, not 15:11
  - RegWrite - all but beq and sw
  - PCSrc - beq taken

Global Control

- Global control outputs
  - Replace PCsrc with
    - Branch beq
    - PCSrc = Branch * Zero
- What are the inputs needed to determine above global control signals?
  - Just Op[5:0]

Global Control

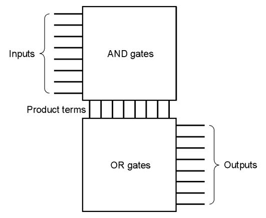
Instruction	Opcode	RegDst	ALUSrc
rrr	000000	1	0
lw	100011	0	1
sw	101011	x	1
beq	000100	x	0
???	others	x	x

- RegDst = ~Op[0]
- ALUSrc = Op[0]
- RegWrite = ~Op[3] * ~Op[2]

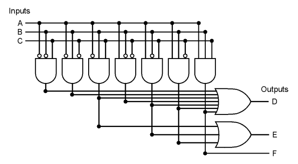
Global Control

- More complex with entire MIPS ISA
  - Need more systematic structure
  - Want to share gates between control signals
- Common solution: PLA
  - MIPS opcode space designed to minimize PLA inputs, minterms, and outputs
- Refer to MIPS Opcode map

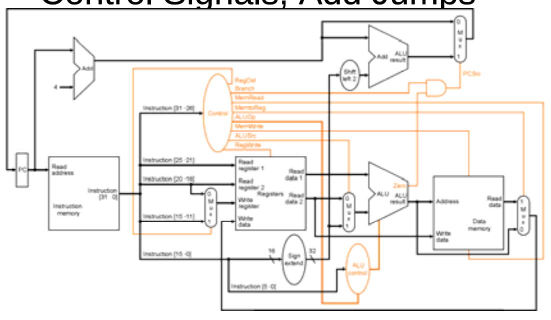
PLA



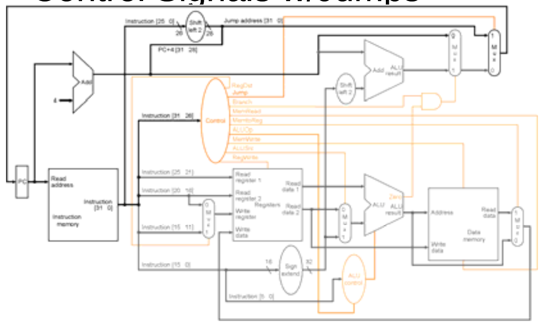
- In AND-plane, & selected inputs to get minterms
- In OR-plane, | selected minterms to get outputs
- E.g.



Control Signals; Add Jumps



Control Signals w/Jumps



What's wrong with single cycle?

$$\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

(code size) **(CPI)** **(cycle time)**

- Critical path probably lw:
  - I-mem, reg-read, alu, d-mem, reg-write
- Other instructions faster
  - E.g. rrr: skip d-mem
- Instruction variation much worse for full ISA and real implementation:
  - FP divide
  - Cache misses (what the heck is this? – later)

Single Cycle Implementation

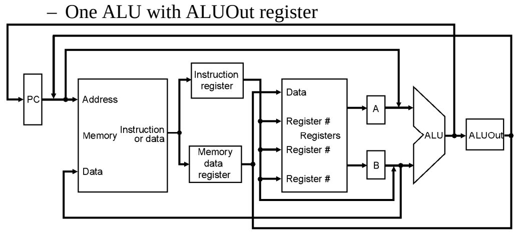
- Solution
  - Variable clock?
    - Too hard to control, design
  - Fixed short clock
    - Variable cycles per instruction

Multi-cycle Implementation

- Clock cycle = max(i-mem,reg-read+reg-write, ALU, d-mem)
- Reuse combinational logic on different cycles
  - One memory
  - One ALU without other adders
- But
  - Control is more complex
  - Need new registers to save values (e.g. IR)
    - Used again on later cycles
    - Logic that computes signals is reused

High-level Multi-cycle Data-path

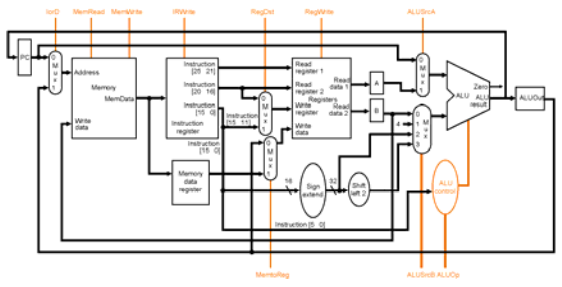
- Note:
  - Instruction register, memory data register
  - One memory with address bus
  - One ALU with ALUOut register



Comment on busses

- Share wires to reduce #signals
  - Distributed multiplexor
- Multiple sources driving one bus
  - Ensure only one is active!

Multi-cycle Ctrl Signals



Multi-cycle Steps

Step	Description	Sample Actions

IF	Fetch	IR=MEM[PC]
		PC=PC+4

ID	Decode	A=RF(IR[25:21])
		B=RF(IR[20:16])
		Target=PC+SE(IR[15:0] << 2)

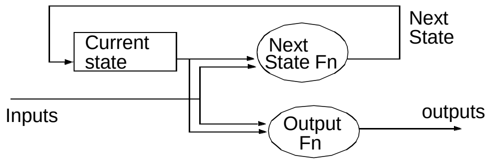
EX	Execute	ALUout = A + SE(IR[15:0]) # lw/sw
		ALUout = A op B # rrr
		if (A==B) PC = target # beq

Mem	Memory	MEM[ALUout] = B # sw
		MDR = MEM[ALUout] #lw
		RF(IR[15:11]) = ALUout # rrr

WB	Writeback	Reg(IR[20:16]) = MDR # lw

Multi-cycle Control

- Function of Op[5:0] and current step
- Defined as Finite State Machine (FSM) or
  - Micro-program or microcode



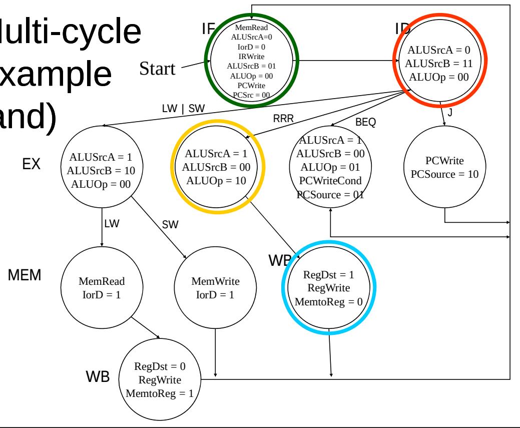
Finite State Machine (FSM)

- For each state, define:
  - Control signals for datapath for this cycle
  - Control signals to determine next state
- All instructions start in same IF state
- Instructions terminate by making IF next
  - After proper PC update, of course

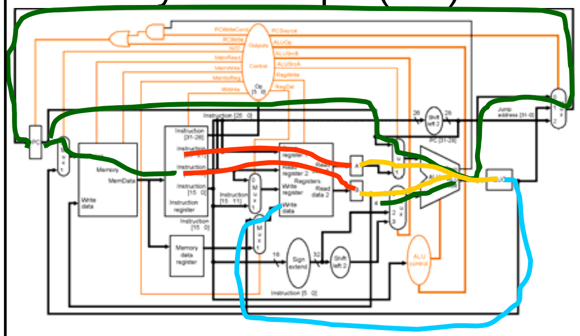
Multi-cycle Example

- Datapath (Fig. 5.33 from book)
  - Will walk *and $1, $2, $3* through datapath
- Look at control FSM (Fig. 5.42 from book)
  - Will walk *and $1, $2, $3* through FSM
- Will skip
  - Repeat for lw, sw, beq taken, beq n-t, j

Multi-cycle Example (and)



Multi-cycle Example (and)



Nuts and Bolts--More on FSMs

- You will be using FSM control for parts of your processor implementation
- There are multiple methods for specifying a state machine
  - Moore machine (output is function of state only)
  - Mealy machine (output is function of state/input)
- There are different methods of assigning states

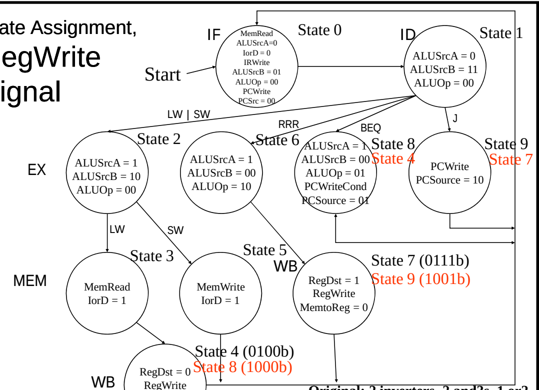
FSMs--State Assignment

- State assignment is converting logical states to binary representation
  - Use ECE 352 methods to build real state machine
- Is state assignment interesting/important?
  - Judicious choice of state representation can make next state fcn/output fcn have fewer gates
  - Optimal solution is hard, but having intuition is helpful (CAD tools can also help in practice)

State Assignment--Example

- 10 states in multicycle control FSM
  - Each state can have 1 of 16 (2^4) encodings with "dense" state representation
  - Any choice of encoding is fine functionally as long as all states are unique
- Appendix C-26 example: RegWrite signal

State Assignment, RegWrite Signal

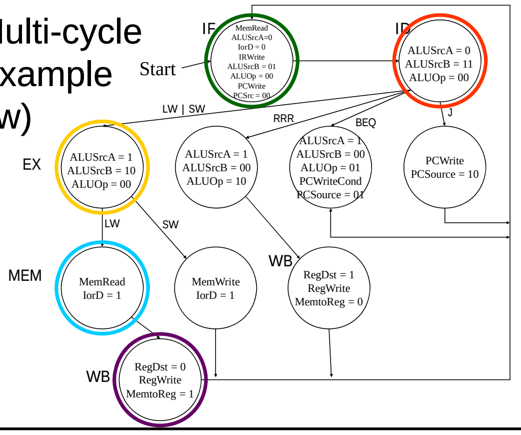


Original: 2 inverters, 2 and3s, 1 or2
New: No gates--just bit 3!

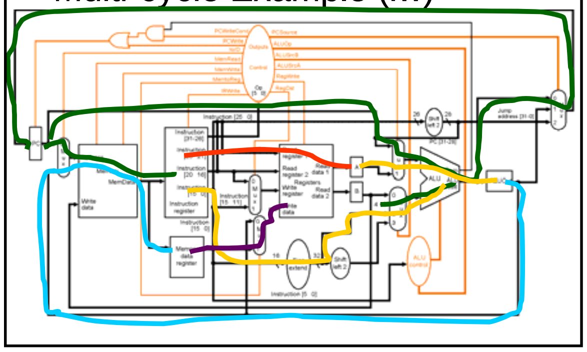
Summary

- Processor implementation
  - Datapath
  - Control
- Single cycle implementation
- Next: microprogramming
- Remaining slides for reference only

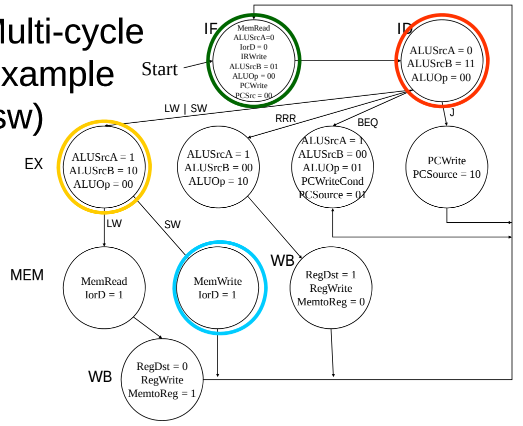
Multi-cycle Example (lw)



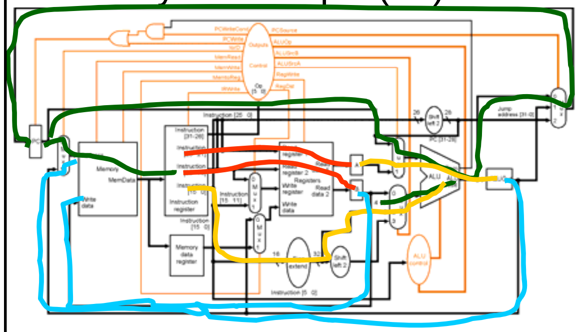
Multi-cycle Example (lw)



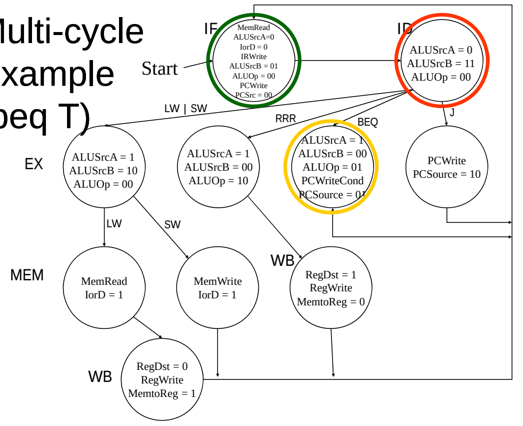
Multi-cycle Example (sw)



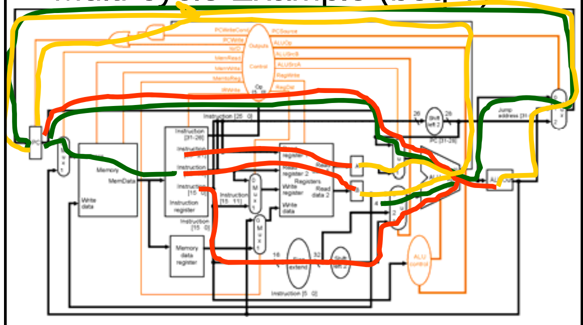
Multi-cycle Example (sw)



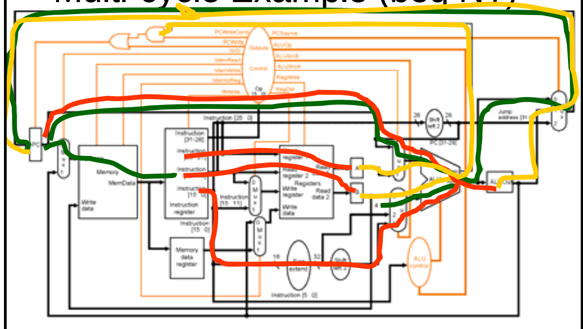
Multi-cycle Example (beq T)



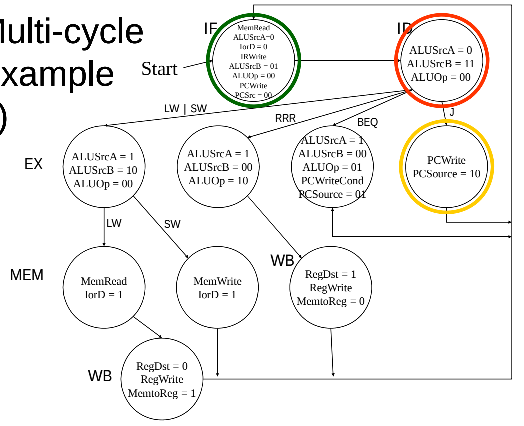
Multi-cycle Example (beq T)



Multi-cycle Example (beq NT)



Multi-cycle Example (j)



Multi-cycle Example (j)