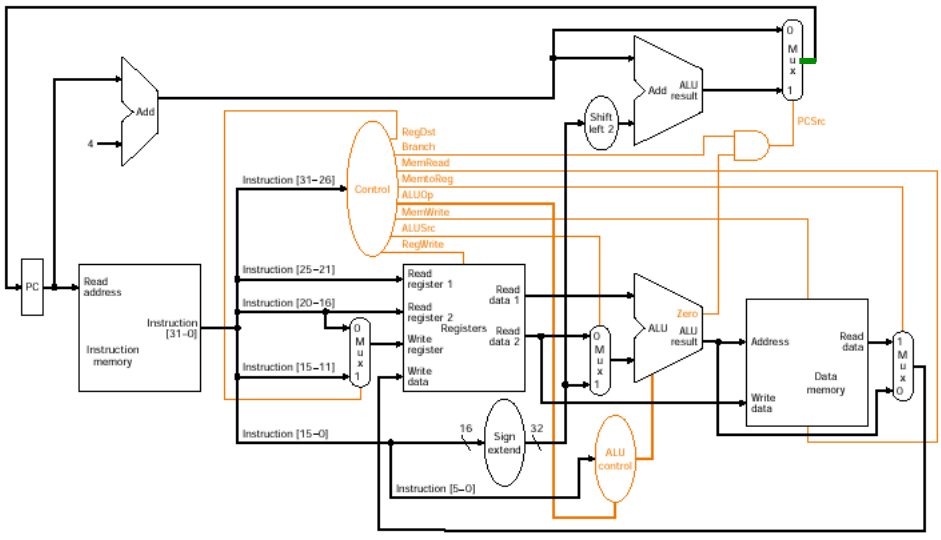
Single vs. Multi-cycle Implementation

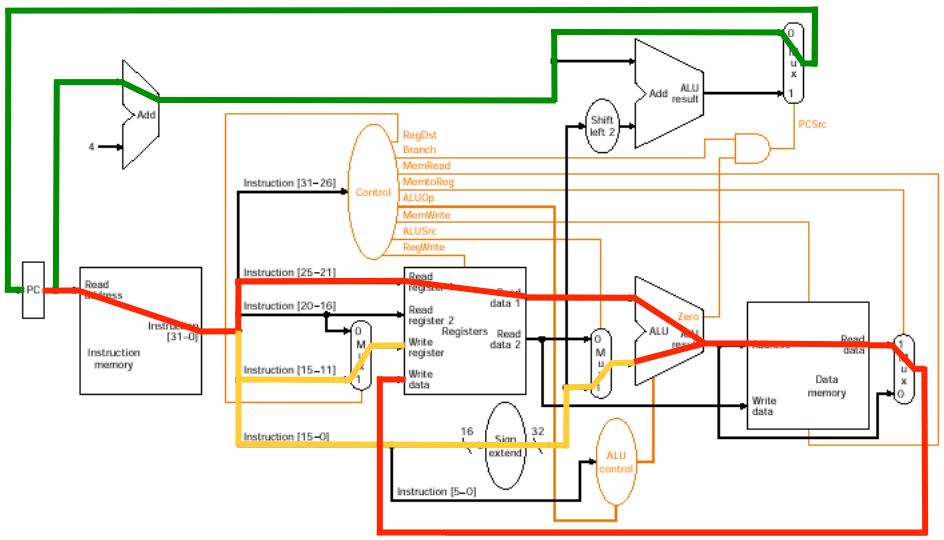
- Single cycle design is simple
- But it's inefficient
- Why?
  - All instructions have same clock cycle length - they all take the same amount of time regardless of what they actually do

- Clock cycle determined by longest path
  - Load: uses IM, RF, ALU, DM, RF in sequence
- But others may be shorter
  - R-type (arithmetic): use IM, RF, ALU, RF

Single Cycle: Load Word



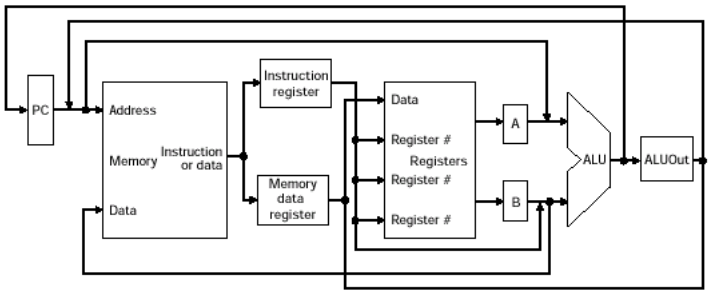
Single Cycle: Load Word



Single vs. Multi-cycle Implementation

- Multicycle: Instructions take *several faster cycles*
- For this simple version, the multi-cycle implementation could be as much as 1.27 times faster (for a typical instruction mix)
- Suppose we had floating point operations
  - Floating point has very high latency
  - E.g., floating-point multiply may be 16 ns vs integer add may be 2 ns
  - So, clock cycle constrained by 16 ns of FP
- Suppose a program doesn't do ANY floating point?
  - Performance penalty is too big to tolerate

Multi-cycle Implementation



- **Single memory unit (I and D), single ALU**
- **Several temporary registers (IR, MDR, A, B, ALUOut)**
- **Temporaries hold output value of element so the output value can be used on subsequent cycle**
- **Values needed by subsequent instruction stored in programmer visible state (memory, RF)**

A single ALU

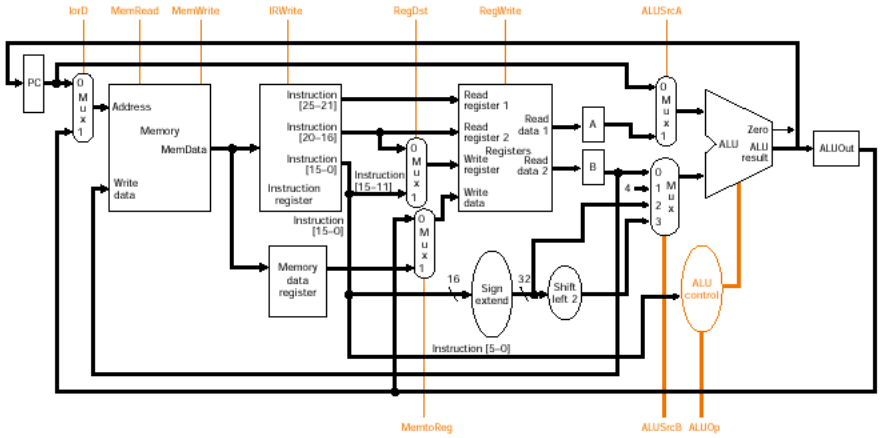
- **Single ALU must accomodate all inputs that used to go to three different ALUs in the single cycle implementation**

1. **Multiplexor on first input to ALU to select A register (from RF) or the PC**

2. **Multiplexor on second input to ALU to select from the constant 4 (PC increment), sign-extended value, shifted offset field, and RF input**

- **Trade-off: Additional multiplexors (and time) but only a single ALU since it can be shared across cycles**

Multi-cycle Datapath with Control



- Datapath with additional muxes, temporary registers, and new control signals
- Most temporaries (except IR) are updated on every cycle, so no write control is required (always write)

Multi-cycle Steps - Instruction Fetch

- Instruction fetch

```
IR = Memory[PC];
PC = PC + 4;
```

- Operation
  - Send PC to memory as the address
  - Read instruction from memory
  - Write instruction into IR for use on next cycle
  - Increment PC by 4
    - Uses ALU in this first cycle
    - Set control signals to send PC and constant 4 to ALU

Multi-cycle Steps - Instruction Decode

- Don't yet know what instruction is
  - Decode the instruction concurrently with RF read
  - Optimistically read registers
  - Optimistically compute branch target
  - We'll select the right answer on next cycle

- Decode and Register File Read

```
A = Reg[IR[25-21]];
B = Reg[IR[20-16]];
ALUOut = PC + (sign-extend(IR[15-0]) << 2);
```


Multi-cycle Steps - Execution

- Operation varies based on instruction decode

- Memory reference:

```
ALUOut = A + sign-extend(IR[15-0]);
```

- Arithmetic-logical instruction:

```
ALUOut = A op B;
```

- Branch:

```
if (A == B) PC = ALUOut;
```

- Jump:

```
PC = PC[31-28] || (IR[25-0] << 2)
```

Multi-cycle Steps - Memory / Completion

- Load/store accesses memory or arithmetic writes result to the register file
- Memory reference:

  `MDR = Memory[ALUOut];` (load)

  or

  `Memory[ALUOut] = B;` (store)
- Arithmetic-logical instruction:

  `Reg[IR[15-11]] = ALUOut;`

Multi-cycle Steps - Read completion

- Finish a memory read by writing read value into the register file
- Load operation:

  `Reg[IR[20-16]] = MDR;`

Multi-cycle Steps

- Instructions always do the first two steps

- Branch can finish in the third step
- Arithmetic-logical can finish in the fourth step
- Stores can finish in the fourth step
- Loads finish in the fifth step

Instruction	Number of cycles
Branch / Jump	3
Arithmetic-logical	4
Stores	4
Loads	5

Multi-cycle vs. Single cycle?

- Why does it help?
- Let's consider a simple example.... in class example

Multi-cycle Instruction Exeution

Branch

```
Cycle0:     IR=Memory[PC];
            PC=PC+4;
Cycle1:     ALUout=PC+(sign-extend(IR[15-0])<<2);
Cycle2:     if A=B PC=ALUout;
```

Arithmetic

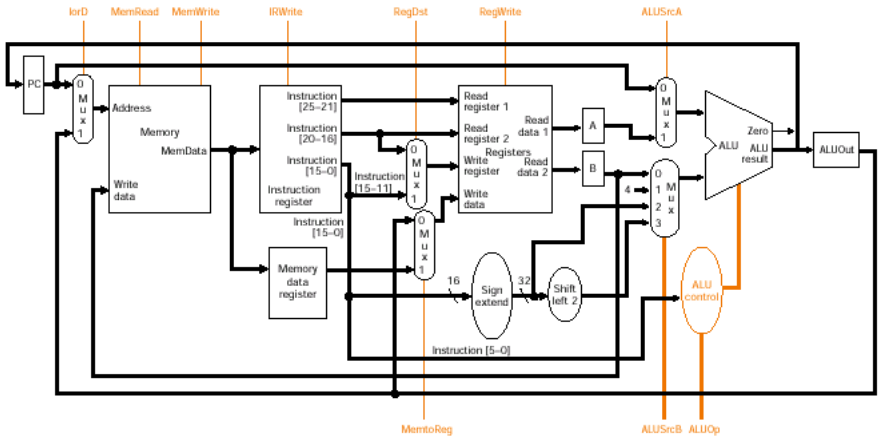
```
Cycle0:     IR=Memory[PC];
            PC=PC+4;
Cycle1:     A=Reg[IR[25-21]]; B=Reg[IR[20-16]];
Cycle2:     ALUout = A op B;
Cycle3:     Reg[IR[15-11]]=ALUout;
```


Multi-cycle Instruction Exeution

Load

```
Cycle0:     IR=Memory[PC];
            PC=PC+4;
Cycle1:     A=Reg[IR[25-21]];
Cycle2:     ALUout = A + sign-extend(IR[15-0]);
Cycle3:     MDR=Memory[ALUout];
Cycle4:     Reg[IR[20-16]]=MDR;
```

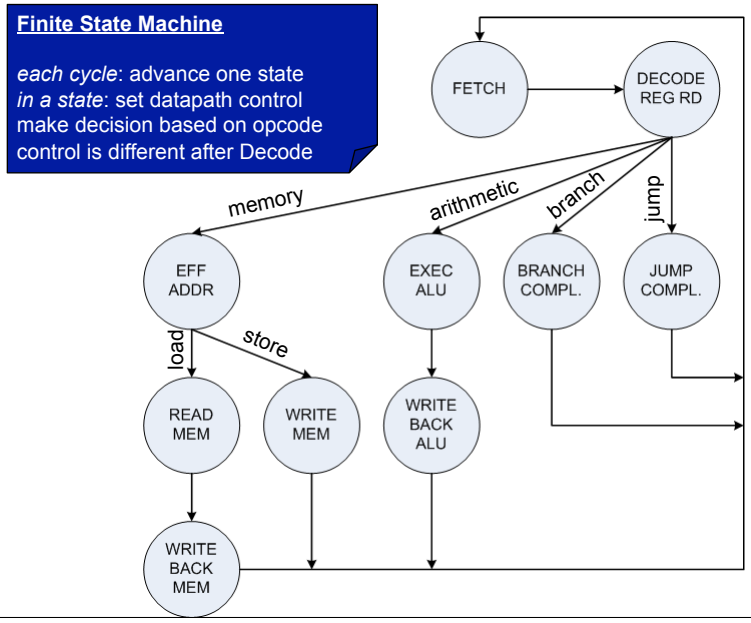
Multi-cycle Datapath with Control



Multi-cycle Control

Finite State Machine

each cycle: advance one state
in a state: set datapath control
make decision based on opcode
control is different after Decode



Multi-cycle Control

- How are the control signals set in each state?
- What are the transitions between states? (i.e., what state is next?)
- Control signals
  - IorD, MemRead, MemWrite, IRWrite, RegDst
  - MemtoReg, RegWrite, ALUSrcA
  - ALUSrcB, ALUOp
  - PCWrite
- Transitions from *Decode* based on Opcode
- Transitions from *Eff. Addr.* happen on load/store

Multi-cycle Control

- What are the control signals in each state for instrs:
  - Arithmetic
  - Load
  - Store
  - Branch
  - Jump

Control for each instruction type?

	STATE (CYCLE NUMBER, ADVANCE EACH CYCLE)				
CONTROL	FETCH(1)	DECODE(2)	STATE 3	STATE 4	STATE 5
IorD					
MemRead					
MemWrite					
IRWrite					
RegDst					
MemToReg					
RegWrite					
ALUSrcA					
ALUSrcB					
ALUOp					
PCWrite					

Control for addition (arithmetic)

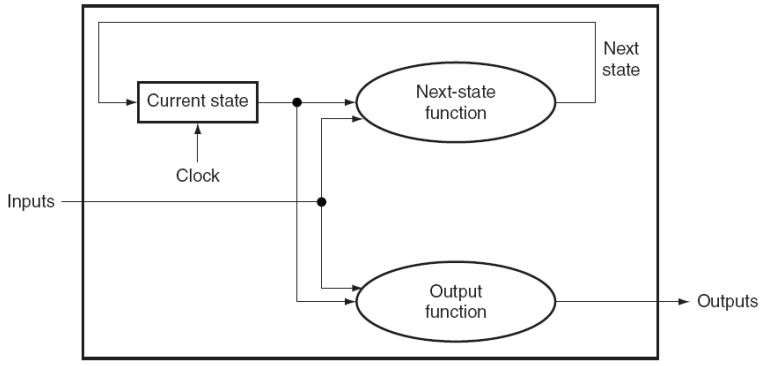
	STATE (CYCLE NUMBER, ADVANCE EACH CYCLE)				
CONTROL	FETCH(1)	DECODE(2)	EXE ALU(3)	WB ALU(4)	STATE 5
IorD	0	X	X	X	
MemRead	1	0	0	0	
MemWrite	0	0	0	0	
IRWrite	1	0	0	0	
RegDst	X	X	X	1	
MemToReg	X	X	X	0	
RegWrite	0	0	0	1	
ALUSrcA	0	0	1	X	
ALUSrcB	01	11	00	X	
ALUOp	00	00	10	X	
PCWrite	1	0	0	0	

Control for addition (load)

	STATE(CYCLE NUMBER, ADVANCE EACH CYCLE)				
CONTROL	FETCH(1)	DECODE(2)	EFF AD(3)	MEM RD(4)	WB MEM(5)
IorD	0	X	X	1	X
MemRead	1	0	0	1	0
MemWrite	0	0	0	0	0
IRWrite	1	0	0	0	0
RegDst	X	X	X	X	0
MemToReg	X	X	X	X	1
RegWrite	0	0	0	0	1
ALUSrcA	0	0	1	X	X
ALUSrcB	01	11	10	X	X
ALUOp	00	00	10	X	X
PCWrite	1	0	0	0	0

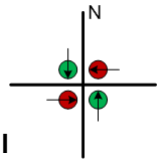
Finite state machine (FSM)

Need a way to specify control per cycle
FSM: Tracks “step of execution” to generate control signals
Implementation: Generally, “**hardwired**” or “microcode”

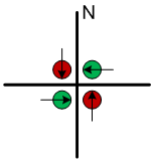


Traffic light control example

- Two states
  - NSgreen: green light on North-South road
  - EWgreen: green light on East-West road

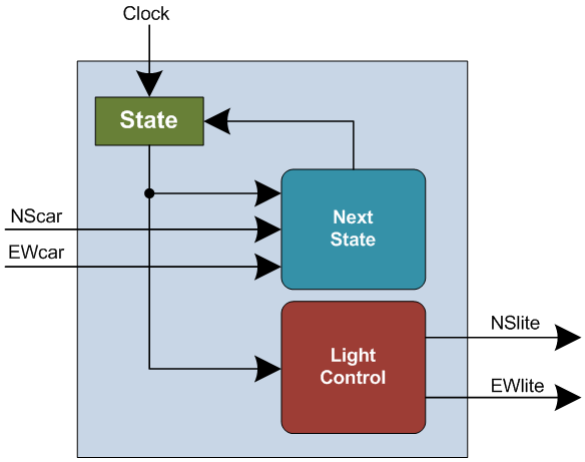


- Sensors (inputs) in each lane to detect car
  - NScar: a car in either the north or south bound l
  - EWcar: a car in either the east or west bound lanes



- Control signals (outputs) to each light
  - NSlite: 0 is red, 1 is green
  - EWlite: 0 is red, 1 is green
- Current state goes for 30 seconds, then
  - Switch to the other state if there is a car waiting
  - Current state goes for another 30 seconds if not
- We use 1/30 Hz clock (Hz is clock cycles per second)
  - I.e., determine a new state (possibly current one) every thirty seconds

Traffic light control example

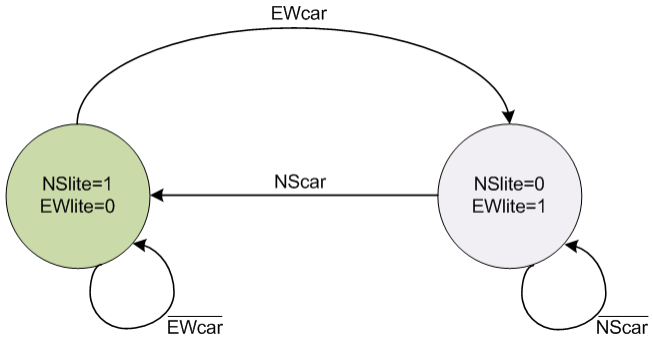


Traffic light control example

Current state	Inputs		Next state
	NScar	EWcar	
NSgreen	0	0	NSgreen
NSgreen	0	1	EWgreen
NSgreen	1	0	NSgreen
NSgreen	1	1	EWgreen
EWgreen	0	0	EWgreen
EWgreen	0	1	EWgreen
EWgreen	1	0	NSgreen
EWgreen	1	1	NSgreen

Current state	Outputs	
	NSlite	EWlite
NSgreen	1	0
EWgreen	0	1

Traffic light control example



Traffic light control example

- Let's assign "0" to NSlite and "1" to EWlite initially
- NextState = CurrentState'·EWcar + CurrentState·NScar'
- NSlite = CurrentState'
- EWlite = CurrentState
- see carfsm.circ on 447 web site