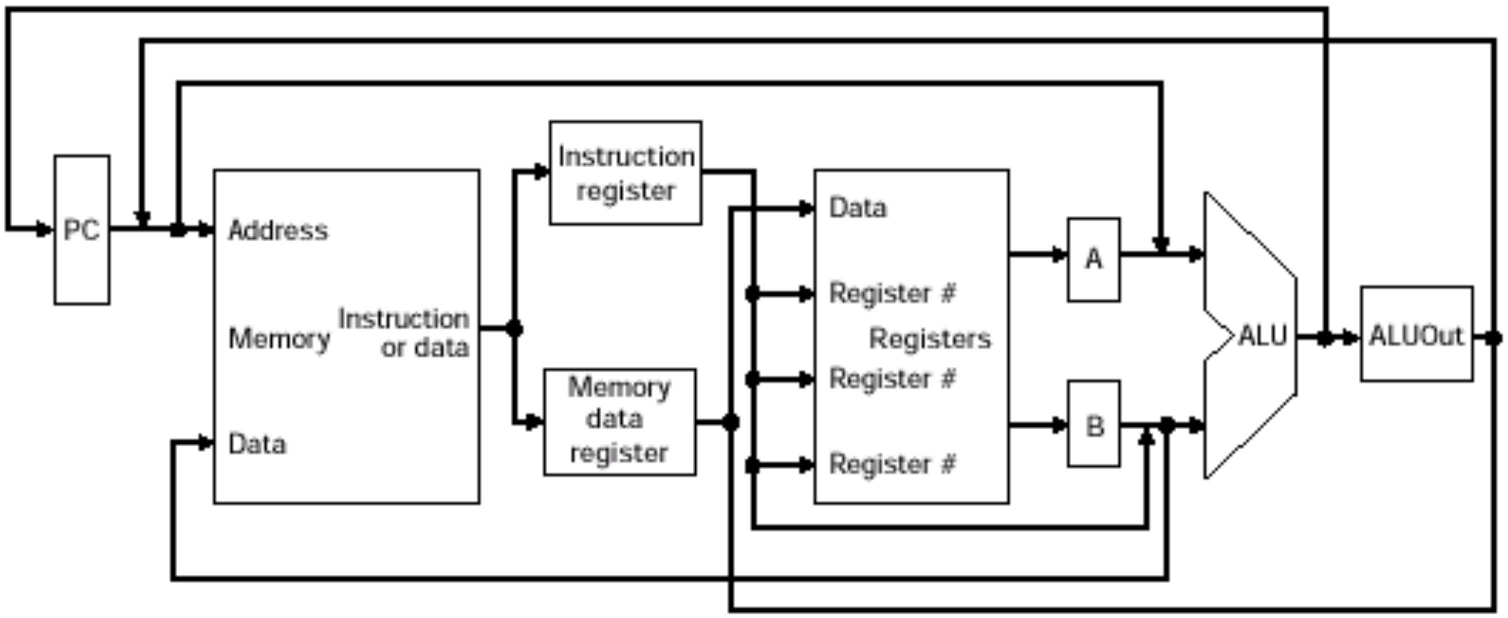
Single vs. Multi-cycle Implementation

- 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)

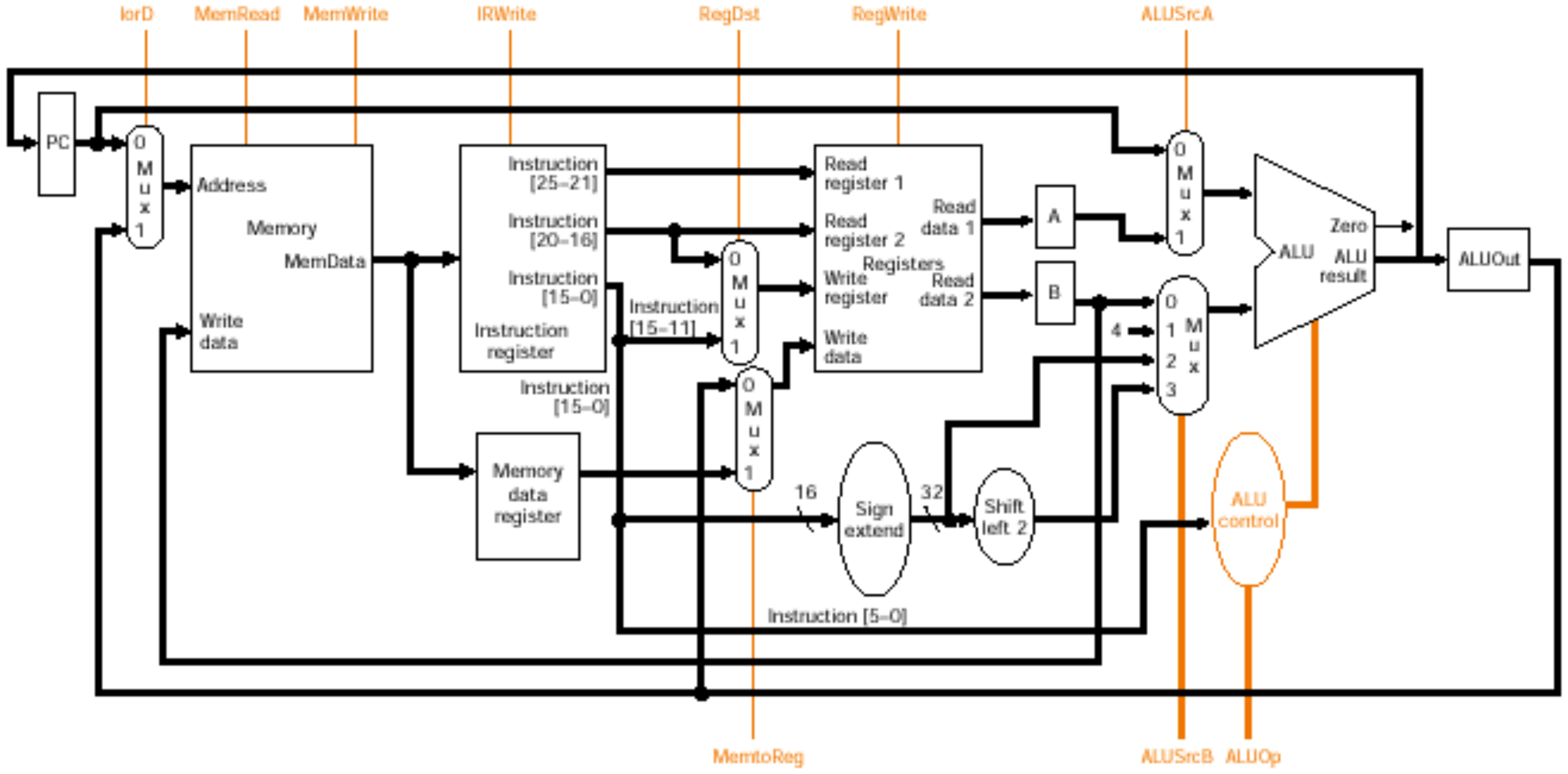
A single ALU

- 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

Example

```
        .data
A:      .word 10,20,30,40,50,60,70,80,90,100
B:      .word 0,0,0,0,0,0,0,0,0,0
        .text
        la      $s0,A           # address of A
        li      $s1,10          # A[i] * 10
        li      $s2,10          # iteration
loop:   lw      $t0,0($s0)      # read A[i]
        mul     $t0,$t0,$s1     # $t0=A[i]*10
        sw      $t0,40($s0)     # write new A[i]
        addi    $s0,$s0,4       # next elemen t
        addi    $s2,$s2,-1      # dec iteration
        bne     $s2,$0,loop     # done?
        li      $v0,10          # exit sys call
        sysall                  # syscall
```

Example

- How much time does it take to execute this program on the single cycle and multi cycle implementation?
- Assume:
  - Single cycle clock rate is 100 MHz

Example

```
        .data
A:      .word 10,20,30,40,50,60,70,80,90,100
B:      .word 0,0,0,0,0,0,0,0,0,0
        .text                 #                   instr. count
        la      $s0,A         # address of A      2
        li      $s1,10        # A[i] * 10         1
        li      $s2,10        # iteration         1
loop:   lw      $t0,0($s0)    # read A[i]         10
        mul     $t0,$t0,$s1   # $t0=A[i]*10       10
        sw      $t0,40($s0)   # write new A[i]    10
        addi    $s0,$s0,4     # next elemen t     10
        addi    $s2,$s2,-1    # dec iteration     10
        bne     $s2,$0,loop   # done?             10
        li      $v0,10        # exit sys call     1
        sysall                # syscall           1
```

Example

- *How much time on SINGLE cycle?*

- Every instruction type takes 1 clock cycle
- Each clock cycle is 100 MHz
- Clock cycle length is 1 / 100 MHz = 10ns

- Sum up the total number of instructions: 66
- Thus,

  66 instruction * 1 cycle each * 10ns per cycle = 660ns

Example

- **How much time on multi cycle?**
- **To answer this, we need to know (1) the clock cycle length for the multi-cycle implementation, and (2) how many instructions of each type are executed**

(1) Suppose ideal circumstance: We divide the single cycle into 5 shorter (faster) cycles:

  - **Multi-cycle clock cycle = 10 ns / 5 cycle= 2 ns**

Example

- (2) How many instructions of each type?
- Easy: Just look at the program and count them

• Arithmetic	36
• Loads	10
• Branches	10
• Stores	10

- Now, we need to compute how much time. The time is simply the sum of the number of each type multiplied by the number of cycles for the type, multiplied by the clock cycle time.

Example

- So, in this case, we have:

  36 arithmetic * 4 cycles * 2 ns +
  10 loads * 5 cycles * 2 ns +
  10 branches * 3 cycles * 2 ns +
  10 stores * 4 cycles * 2 ns
  = 288 ns + 100 ns + 60 ns + 80 ns
  = 528 ns

- Thus, multi-cycle implementation is 528 ns and the single cycle is 660 ns.

Example

- How much faster is the multi-cycle implementation?
- Ratio of time for single cycle to multi cycle
- Thus, we have:

  660 ns / 528 ns = 1.25 times faster

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

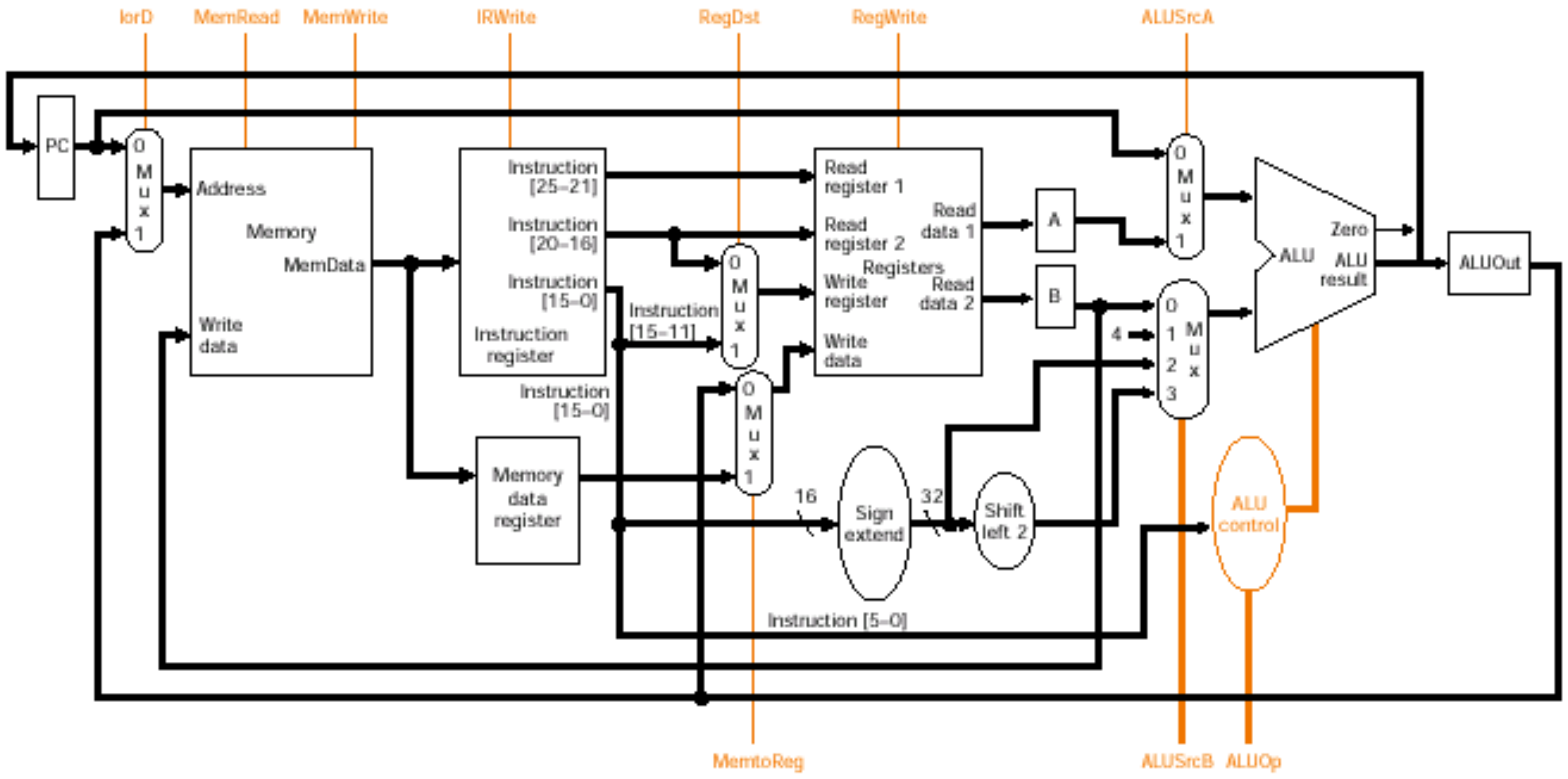
Load

```
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 Control

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

Multi-cycle Datapath with Control



Multi-cycle Control

- How are *control signals generated* on each cycle?
- What are the *transitions* between cycles? (i.e., what happens next?)

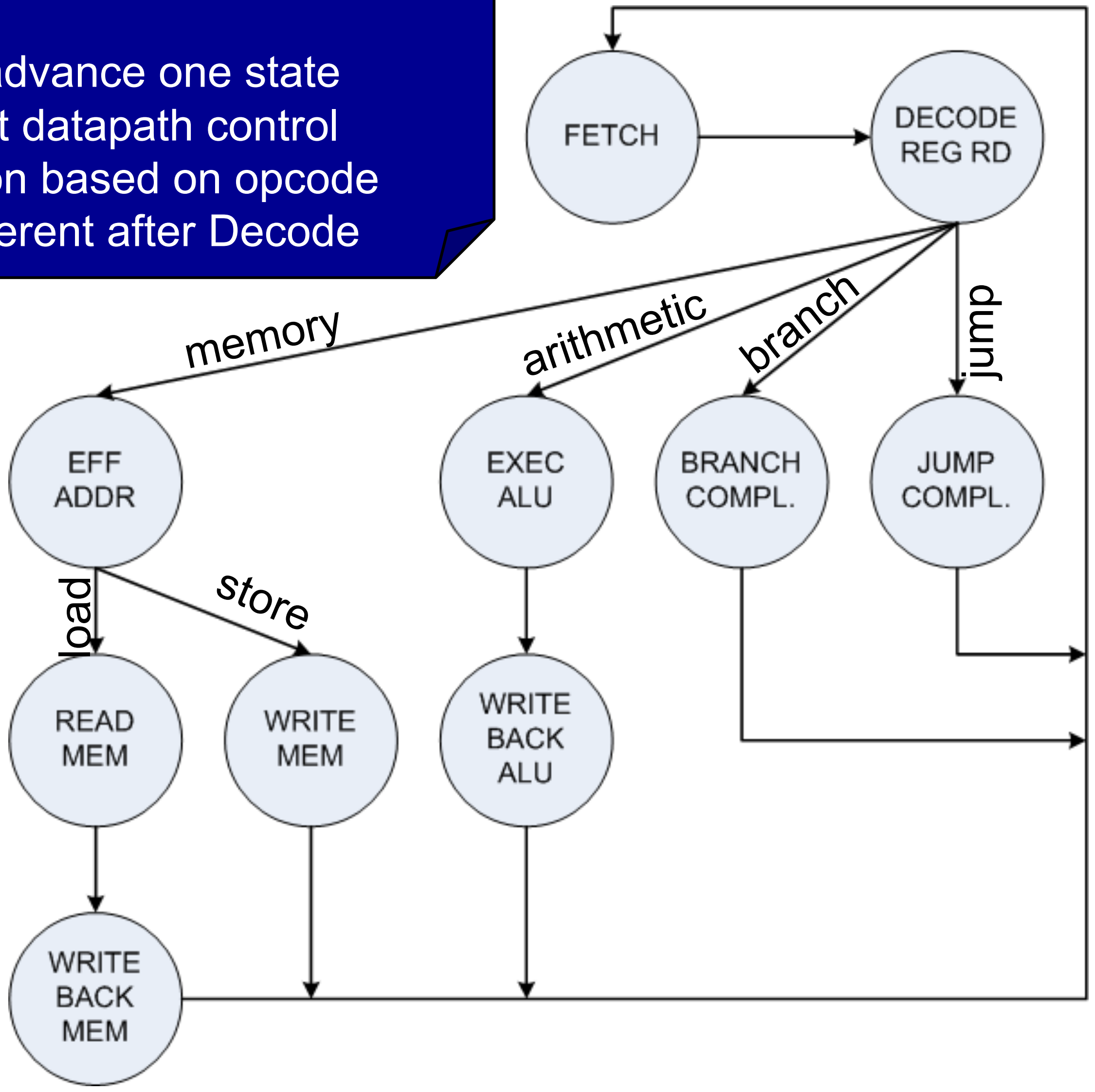
- Control signals
  - IorD, MemRead, MemWrite, IRWrite, RegDst
  - MemtoReg, RegWrite, ALUSrcA
  - ALUSrcB, ALUOp
  - PCWrite

- Transitions from *Decode* based on Opcode
- Transitions from *Eff. Addr.* happen on load/store

Multi-cycle Control

Finite State Machine

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



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 memory (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

Control for memory (store)

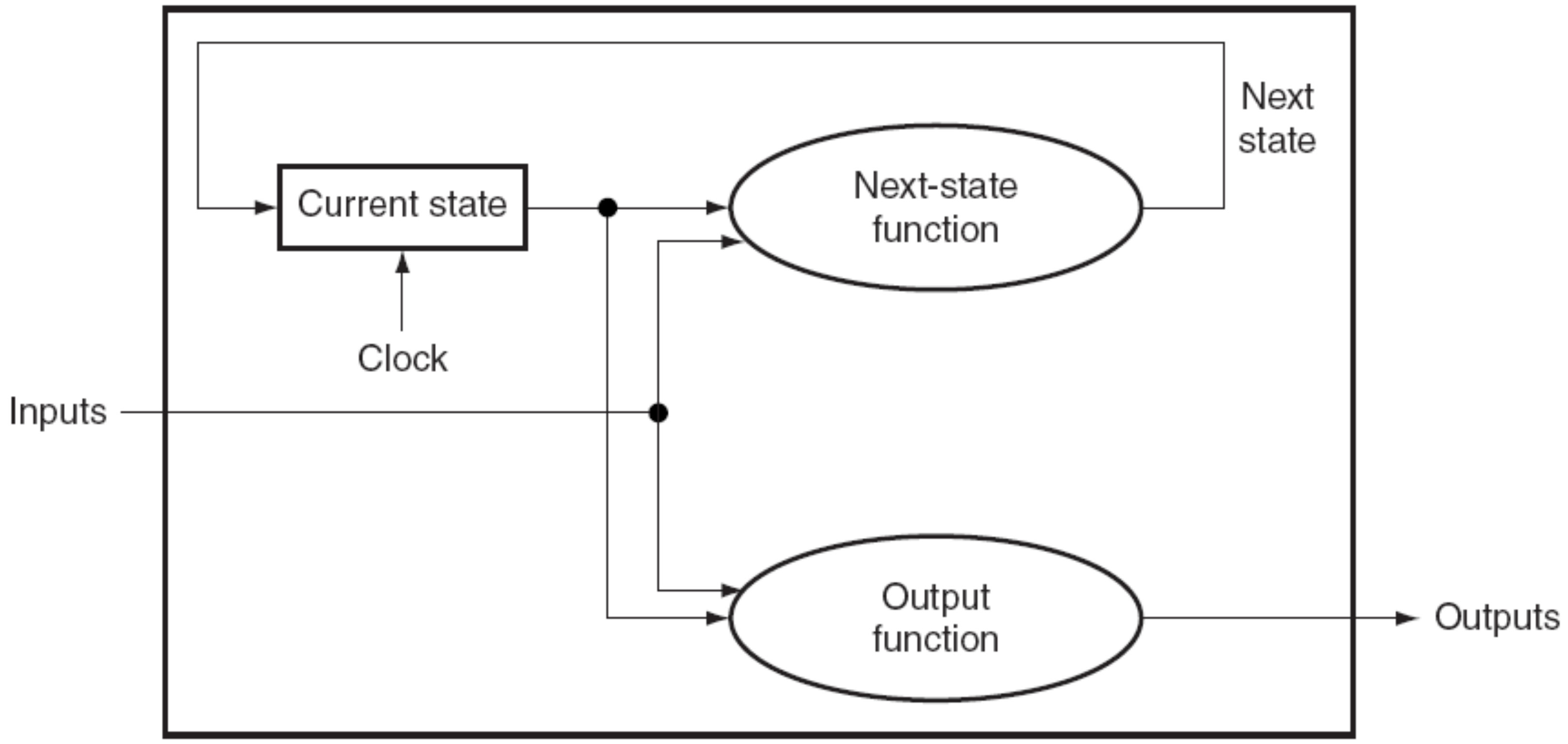
	STATE(CYCLE NUMBER, ADVANCE EACH CYCLE)				
CONTROL	FETCH(1)	DECODE(2)	EFF AD(3)	MEM WR(4)	
IorD	0	X	X	1	
MemRead	1	0	0	0	
MemWrite	0	0	0	1	
IRWrite	1	0	0	0	
RegDst	X	X	X	X	
MemToReg	X	X	X	X	
RegWrite	0	0	0	0	
ALUSrcA	0	0	1	X	
ALUSrcB	01	11	10	X	
ALUOp	00	00	10	X	
PCWrite	1	0	0	0	

Control for branch

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

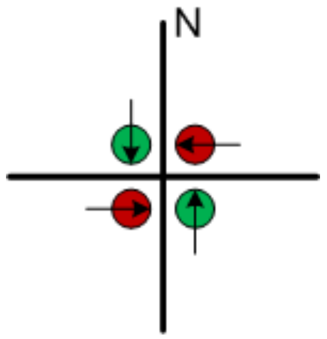
Finite state machine (FSM)

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



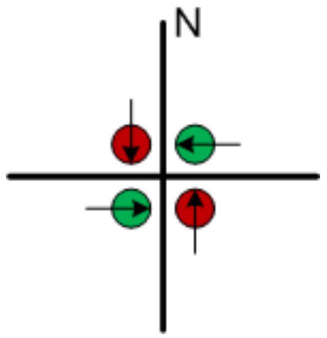
Traffic light control example

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



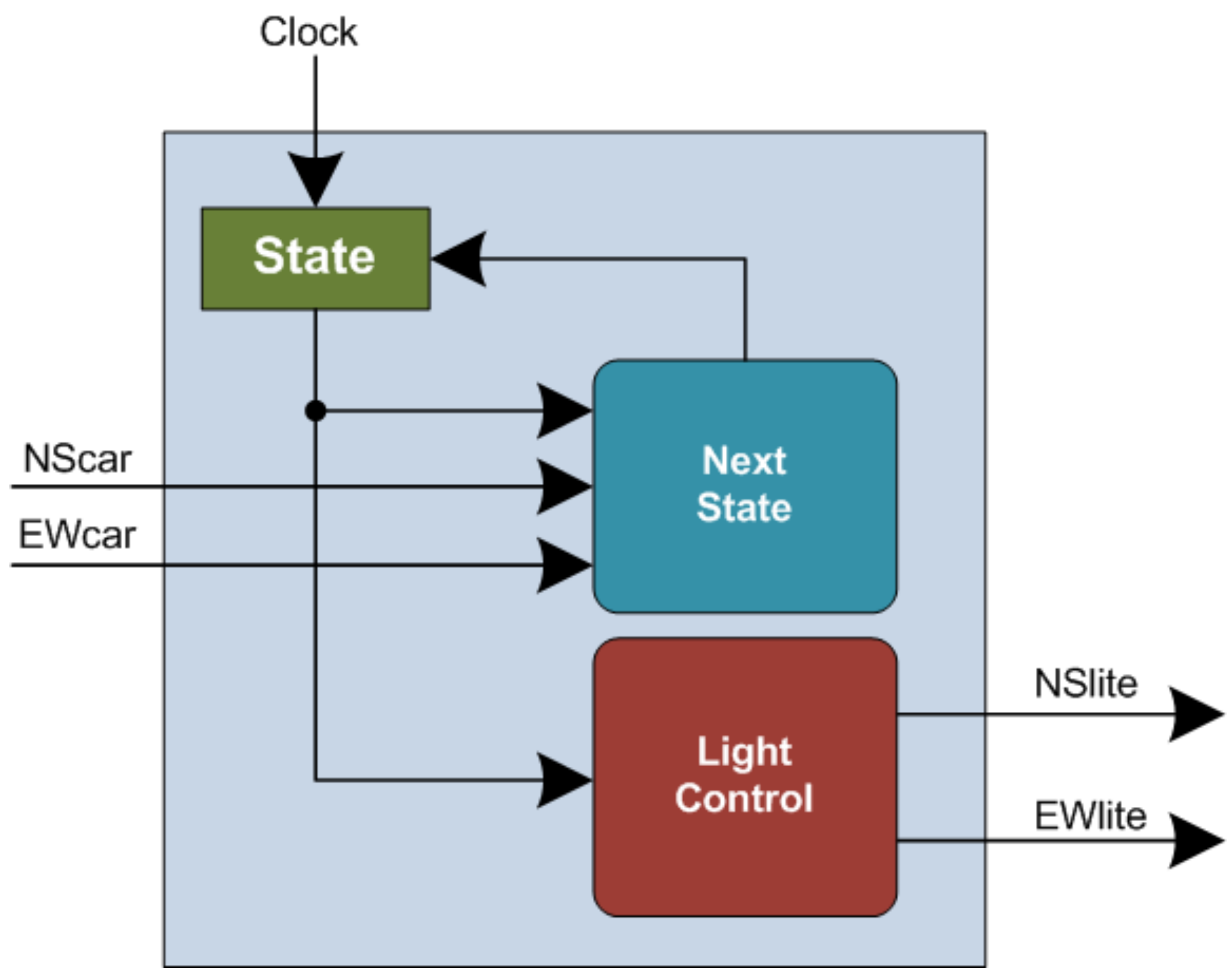
- Sensors (inputs) in each lane to detect car
  - NScar: a car in either the north or south bound lanes
  - EWcar: a car in either the east or west bound lanes

- 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

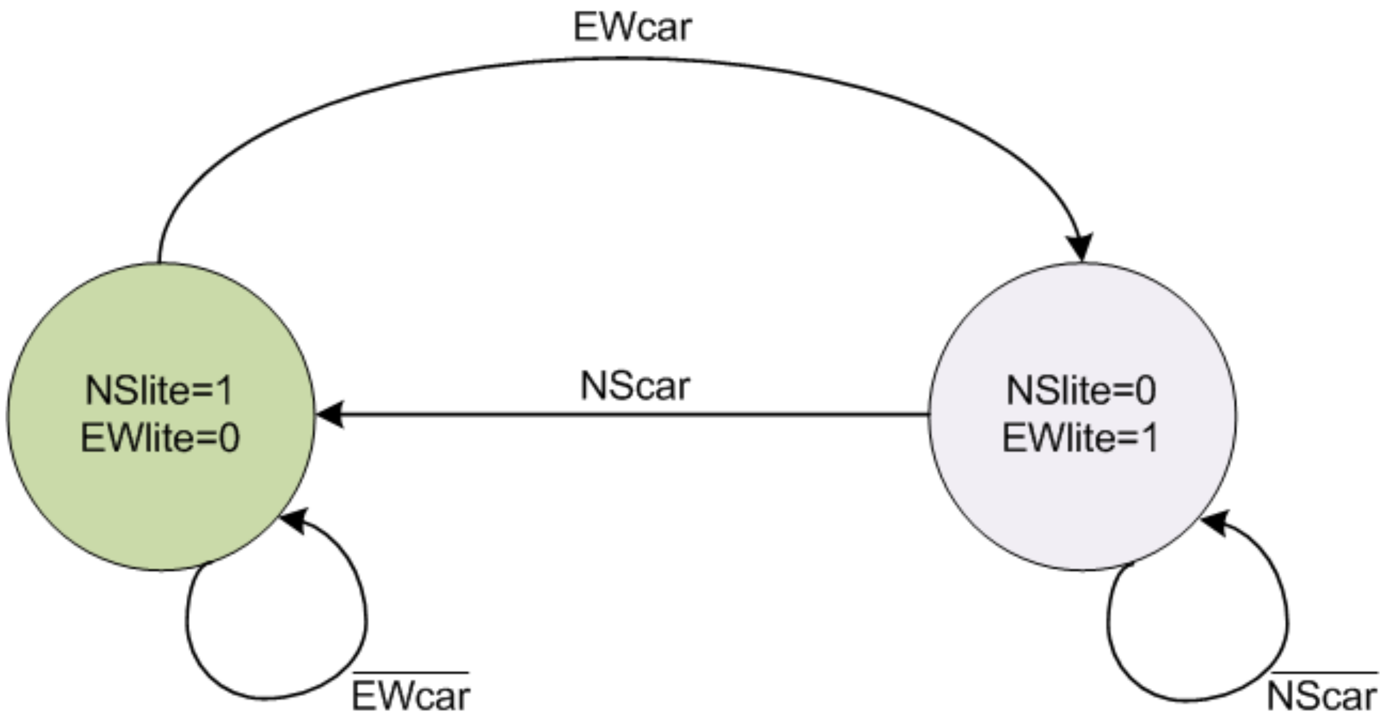


Traffic light control example

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