Single Cycle Implementation Cycle Time

- Unfortunately, though simple, the single cycle approach is not used because it is very slow
- Clock cycle must have the same length for every instruction
- What is the longest (slowest) path (slowest instruction)?

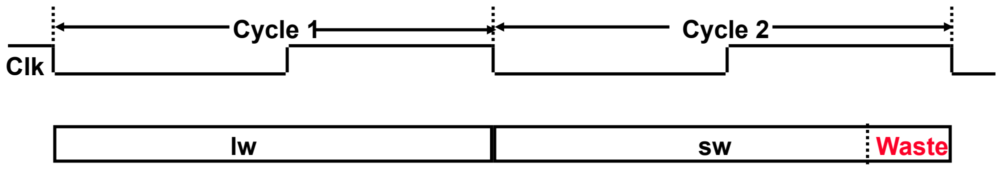
Instruction Critical Paths

- Calculate cycle time assuming negligible delays (for muxes, control unit, sign extend, PC access, shift left 2, wires, setup and hold times) except:
  - Instruction and Data Memory (200 ps)
  - ALU and adders (100 ps)
  - Register File access (reads or writes) (100 ps)

Instr.	I Mem	Reg Rd	ALU Op	D Mem	Reg Wr	Total
R-type	200	100	100		100	500
load	200	100	100	200	100	700
store	200	100	100	200		600
beq	200	100	100			400
jump	200					200

Single Cycle Disadvantages & Advantages

- Uses the clock cycle inefficiently – the clock cycle must be timed to accommodate the slowest instr
  - especially problematic for more complex instructions like floating point multiply



- May be wasteful of area since some functional units (e.g., adders) must be duplicated since they can not be shared during a clock cycle

but

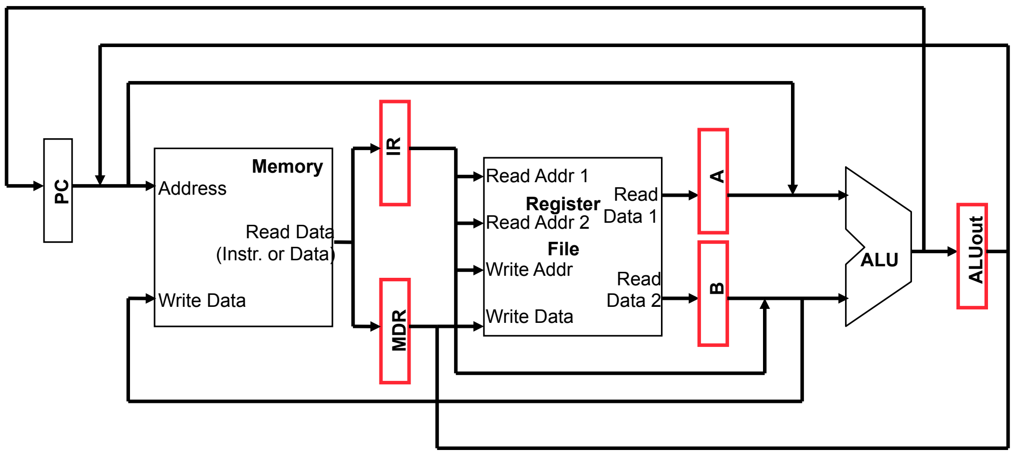
- It is simple and easy to understand

Multicycle Implementation Overview

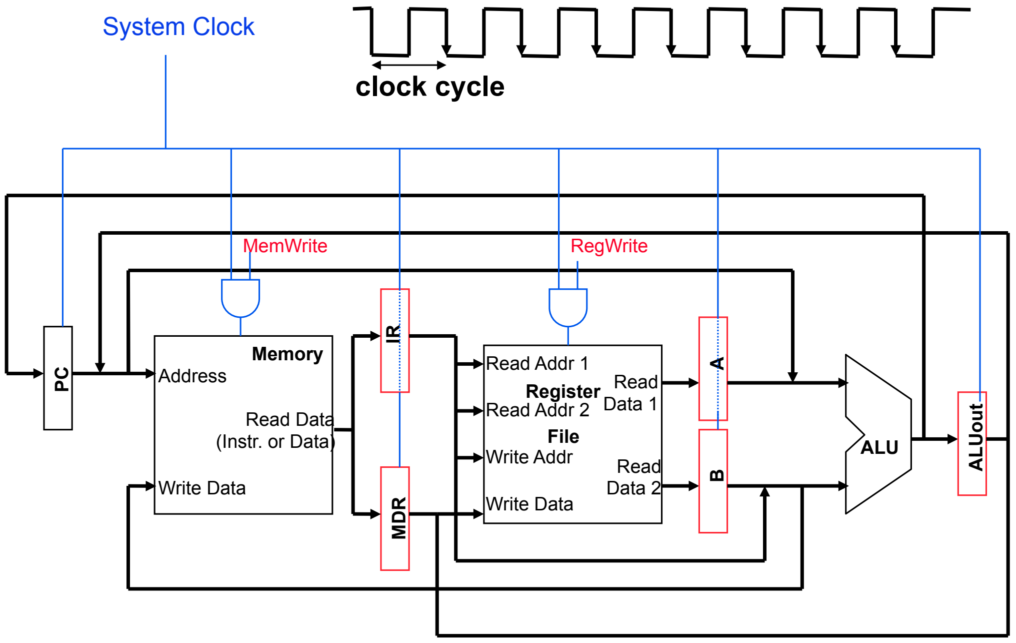
- Each instruction step takes 1 clock cycle
  - Therefore, an instruction takes more than 1 clock cycle to complete
- Not every instruction takes the same number of clock cycles to complete
- Multicycle implementations allow
  - faster clock rates
  - different instructions to take a different number of clock cycles
  - functional units to be used more than once per instruction as long as they are used on different clock cycles, as a result
    - only need one memory
    - only need one ALU/adder

The Multicycle Datapath – A High Level View

- Registers have to be added after every major functional unit to hold the output value until it is used in a subsequent clock cycle



Clocking the Multicycle Datapath



Our Multicycle Approach

- ❑ Break up the instructions into steps where each step takes a clock cycle while trying to
  - balance the amount of work to be done in each step
  - use only one major functional unit per clock cycle
- ❑ At the end of a clock cycle
  - Store values needed in a later clock cycle by the current instruction in a state element (internal register not visible to the programmer)

    IR – Instruction Register

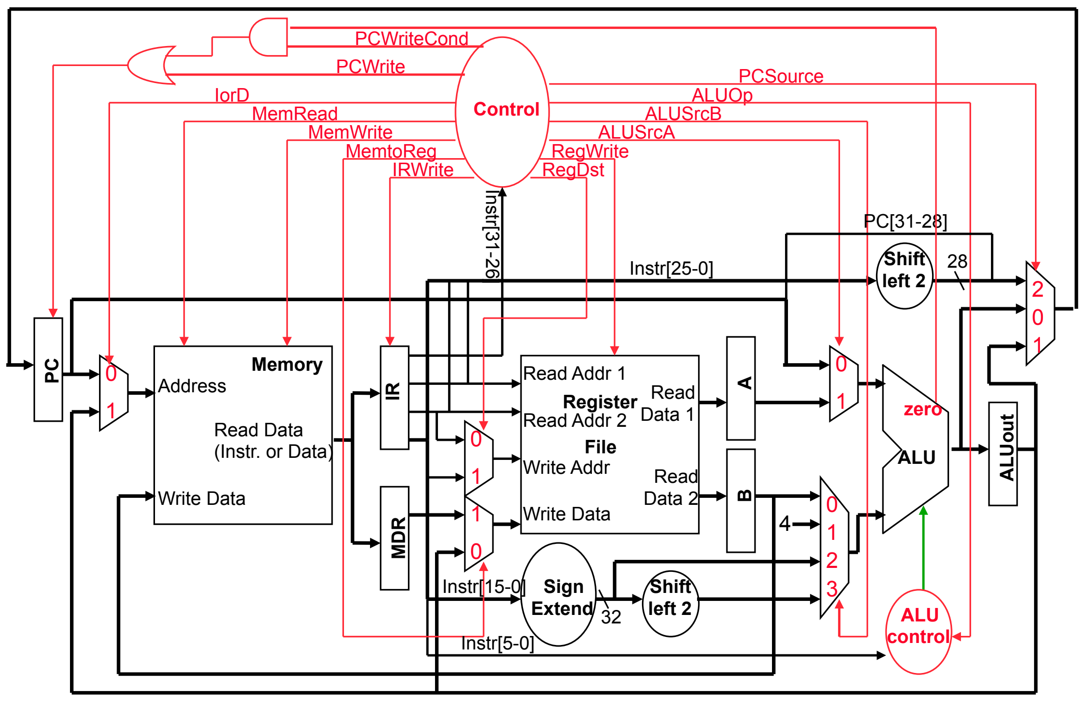
    MDR – Memory Data Register

    A and B – Register File read data registers

    ALUout – ALU output register

    - All (except IR) hold data only between a pair of adjacent clock cycles (so they don't need a write control signal)
  - Data used by subsequent instructions are stored in programmer visible state elements (i.e., Register File, PC, or Memory)

The Complete Multicycle Data with Control



Review: Our ALU Control

- Controlling the ALU uses of multiple decoding levels
  - main control unit generates the ALUOp bits
  - ALU control unit generates ALUcontrol bits

Instr op	funct	ALUOp	action	ALUcontrol
lw	xxxxxx	00	add	0110
sw	xxxxxx	00	add	0110
beq	xxxxxx	01	subtract	1110
add	100000	10	add	0110
subt	100010	10	subtract	1110
and	100100	10	and	0000
or	100101	10	or	0001
xor	100110	10	xor	0010
nor	100111	10	nor	0011
slt	101010	10	slt	1111

Our Multicycle Approach, con't

- Reading from or writing to any of the internal registers, Register File, or the PC occurs (quickly) at the beginning (for read) or the end of a clock cycle (for write)
- Reading from the Register File takes ~50% of a clock cycle since it has additional control and access overhead (but reading can be done in parallel with decode)
- Had to add multiplexors in front of several of the functional unit input ports (e.g., Memory, ALU) because they are now shared by different clock cycles and/or do multiple jobs
- All operations occurring in one clock cycle occur in parallel
  - This limits us to one ALU operation, one Memory access, and one Register File access per clock cycle

Five Instruction Steps

1. Instruction Fetch
2. Instruction Decode and Register Fetch
3. R-type Instruction Execution, Memory Read/Write Address Computation, Branch Completion, or Jump Completion
4. Memory Read Access, Memory Write Completion or R-type Instruction Completion
5. Memory Read Completion (Write Back)

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

Step 1: Instruction Fetch

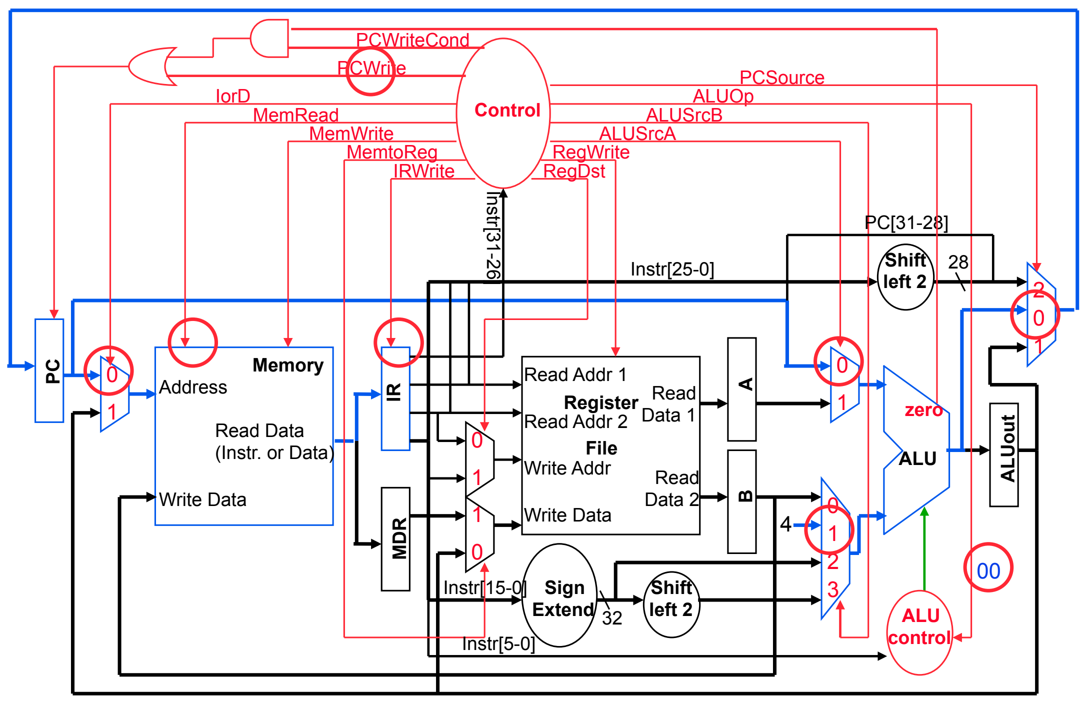
- Use PC to get instruction from the memory and put it in the Instruction Register
- Increment the PC by 4 and put the result back in the PC
- Can be described succinctly using the RTL "Register-Transfer Language“

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

Can we figure out the values of the control signals?

What is the advantage of updating the PC now?

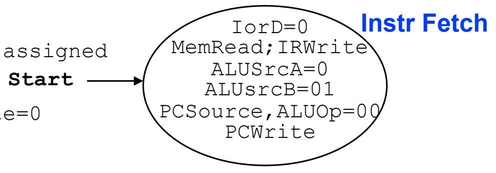
Datapath Activity During Instruction Fetch



Fetch Control Signals Settings

```
Unless otherwise assigned

 PCWrite,IRWrite,
 MemWrite,RegWrite=0
 others=X
```



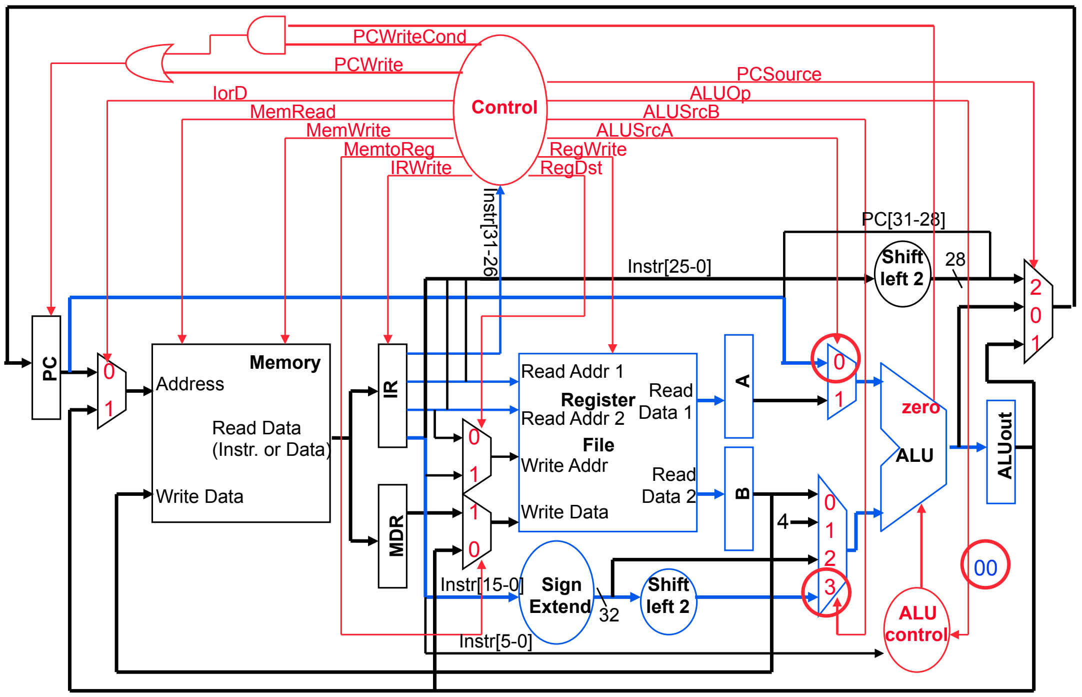
Step 2: Instruction Decode and Register Fetch

- ❑ Don't know what the instruction is yet, so can only
  - Read registers rs and rt in case we need them
  - Compute the branch address in case the instruction is a branch
- ❑ The RTL:

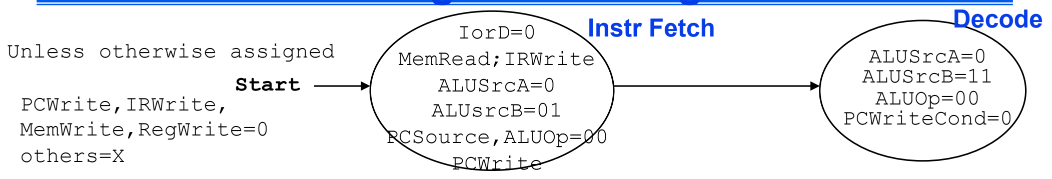
```
 A = Reg[IR[25-21]];
B = Reg[IR[20-16]];
ALUOut = PC
           +(sign-extend(IR[15-0])<< 2);
```

- ❑ Note we aren't setting any control lines based on the instruction (since we don't know what it is (the control logic is busy "decoding" the op code bits))

Datapath Activity During Instruction Decode



Decode Control Signals Settings



Step 3 (instruction dependent)

- ALU is performing one of four functions, based on instruction type
- Memory reference (`lw` and `sw`):

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

- R-type:

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

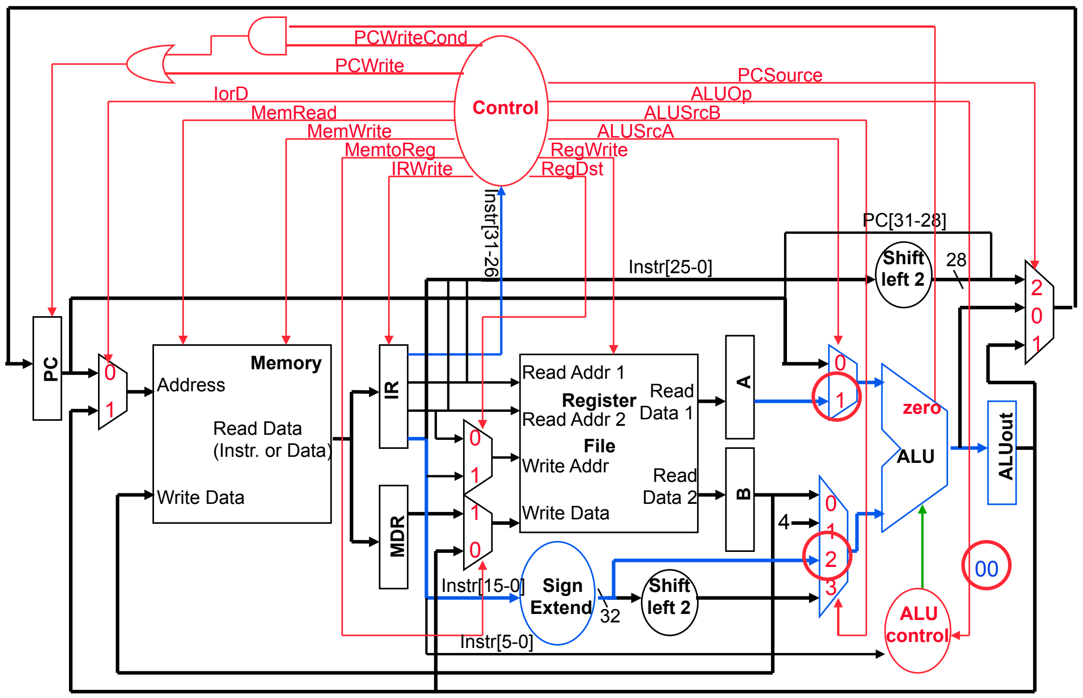
- Branch:

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

- Jump:

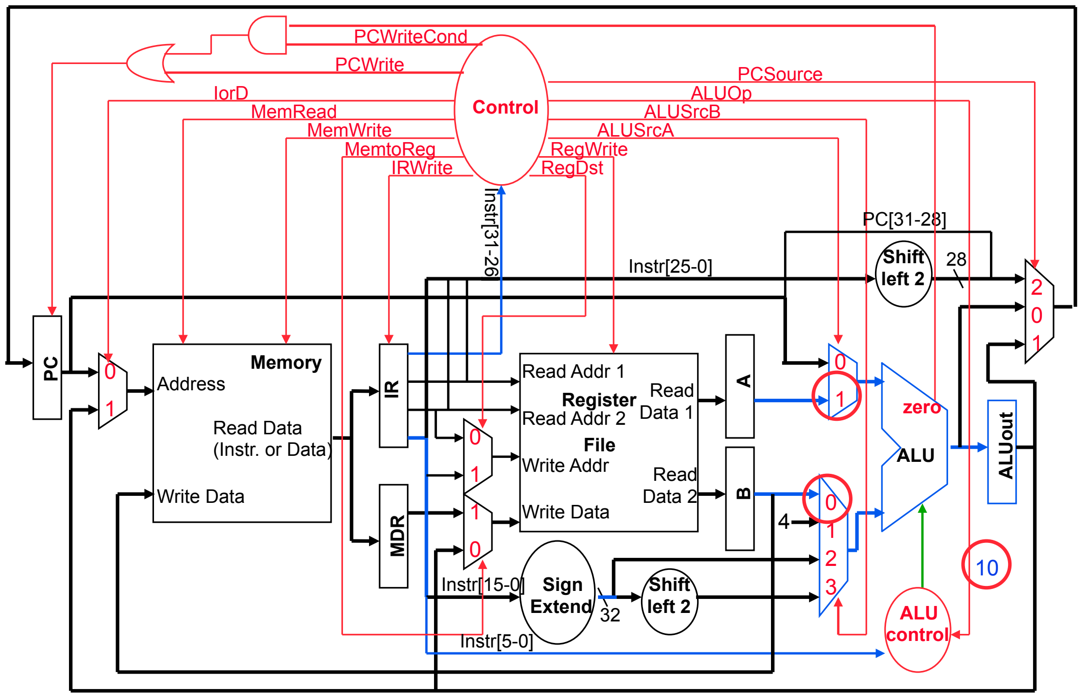
```
    PC = PC[31-28] || (IR[25-0] << 2);
```

Datapath Activity During `lw` & `sw` Execute



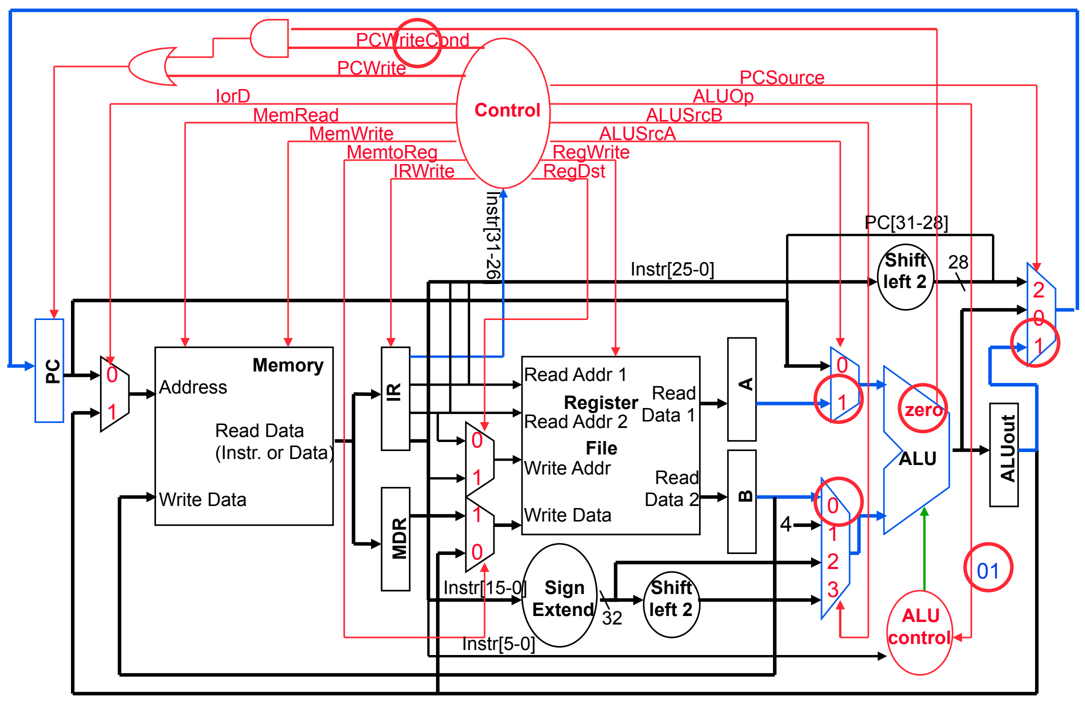
Bressoud Spring 2010

Datapath Activity During R-type Execute

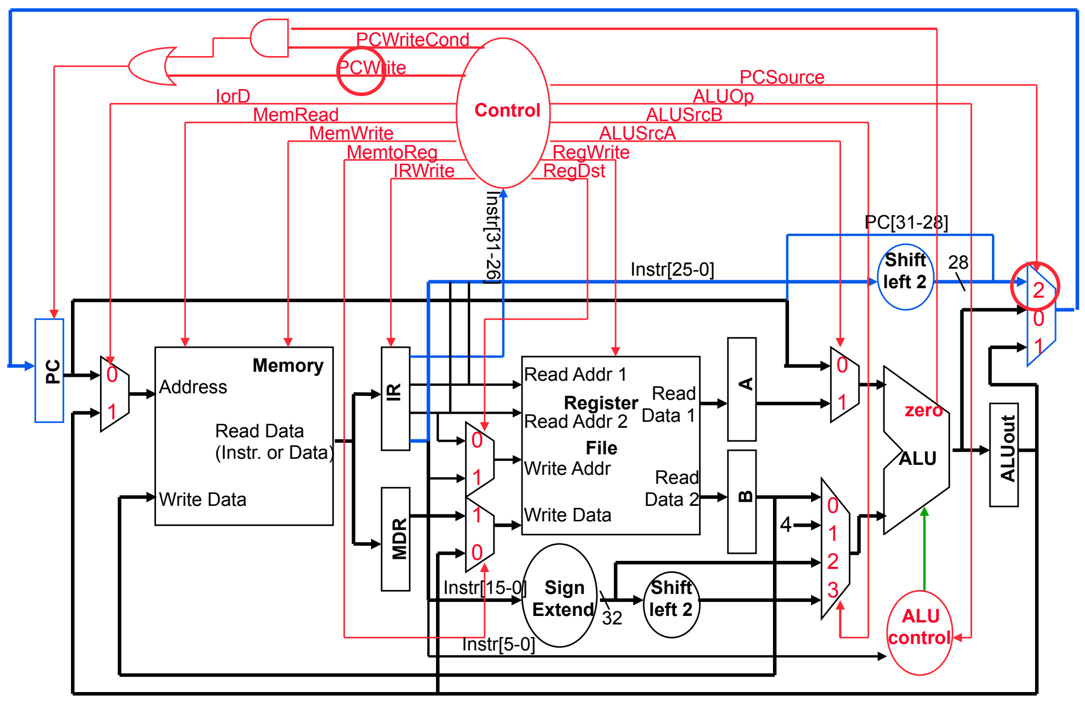


Bressoud Spring 2010

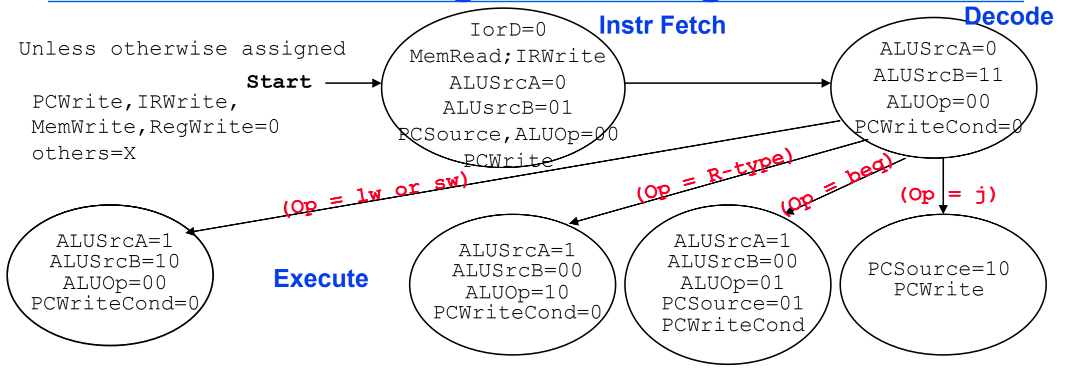
Datapath Activity During beq Execute



Datapath Activity During j Execute



Execute Control Signals Settings


Unless otherwise assigned

PCWrite,IRWrite,
MemWrite,RegWrite=0
others=X

Step 4 (also instruction dependent)

- Memory reference:

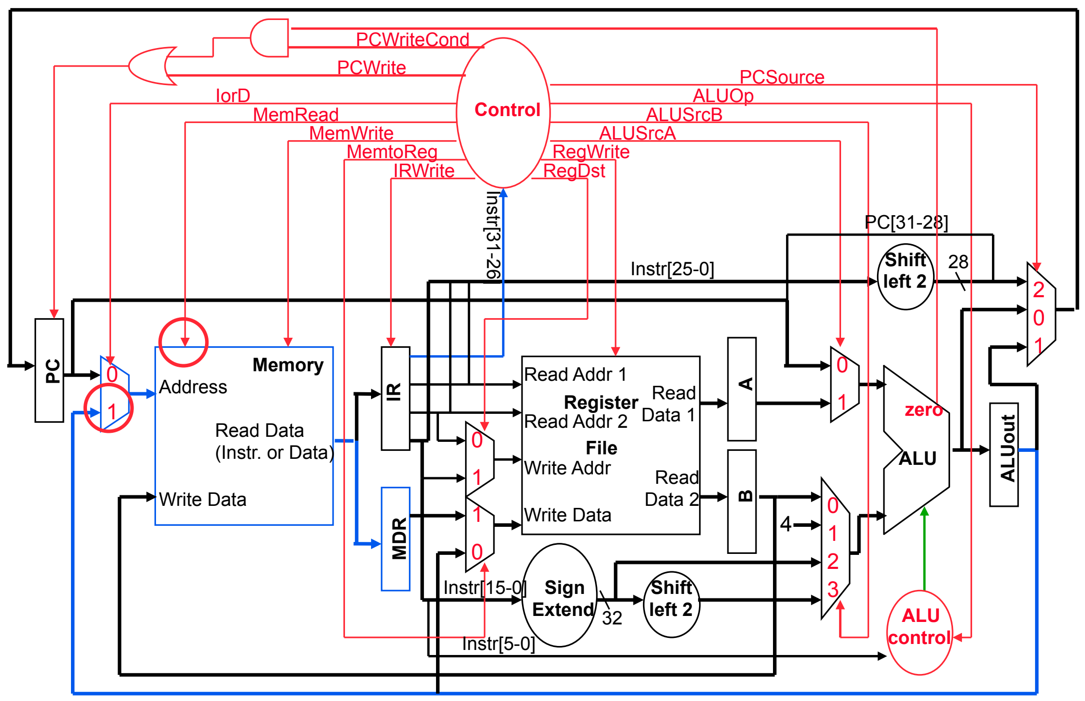
```
    MDR = Memory[ALUOut];      -- lw
or
    Memory[ALUOut] = B;        -- sw
```

- R-type instruction completion

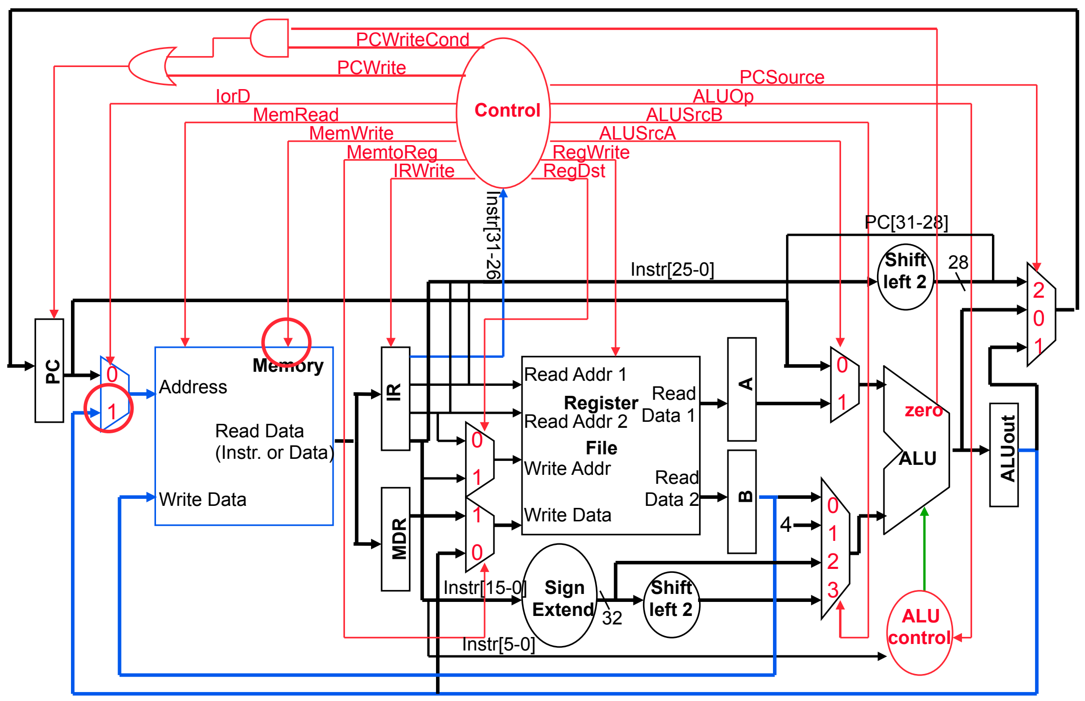
```
    Reg[IR[15-11]] = ALUOut;
```

- Remember, the register write actually takes place at the end of the cycle on the clock edge

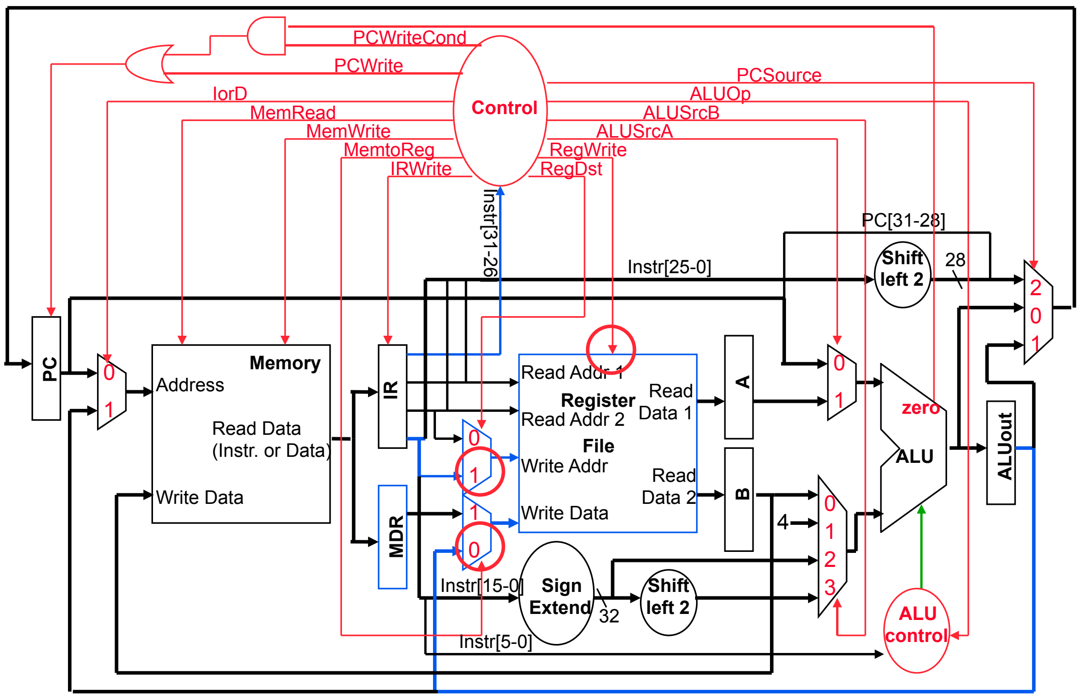
Datapath Activity During `lw` Memory Access



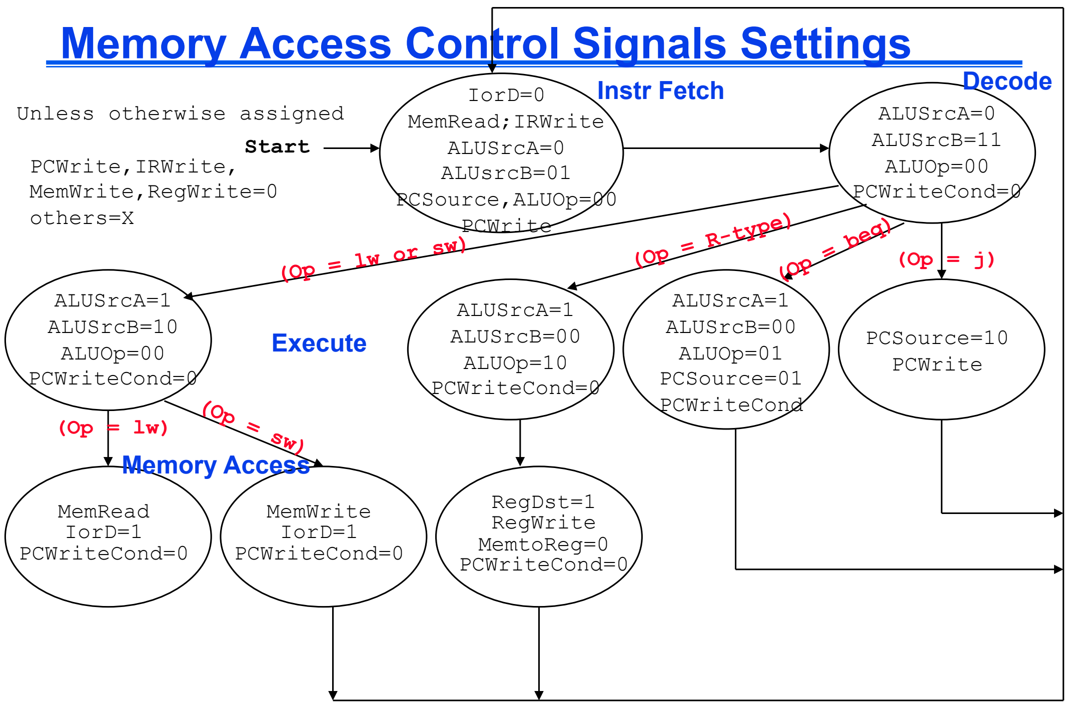
Datapath Activity During `sw` Memory Access



Datapath Activity During R-type Completion



Memory Access Control Signals Settings



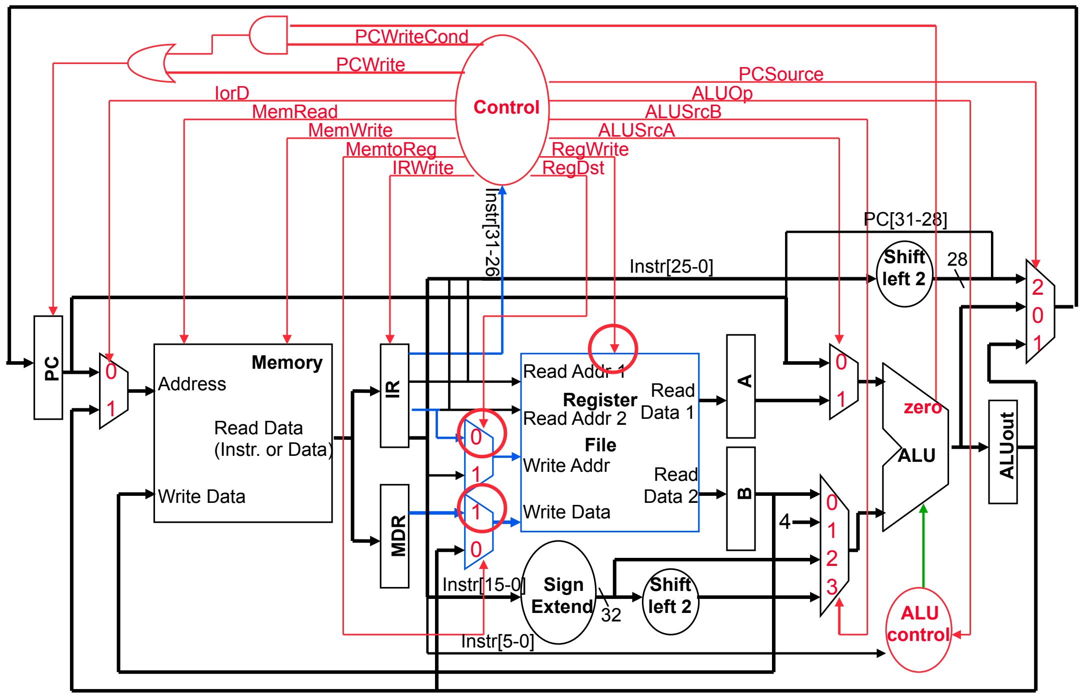
Step 5: Memory Read Completion (Write Back)

- All we have left is the write back into the register file the data just read from memory for lw instruction

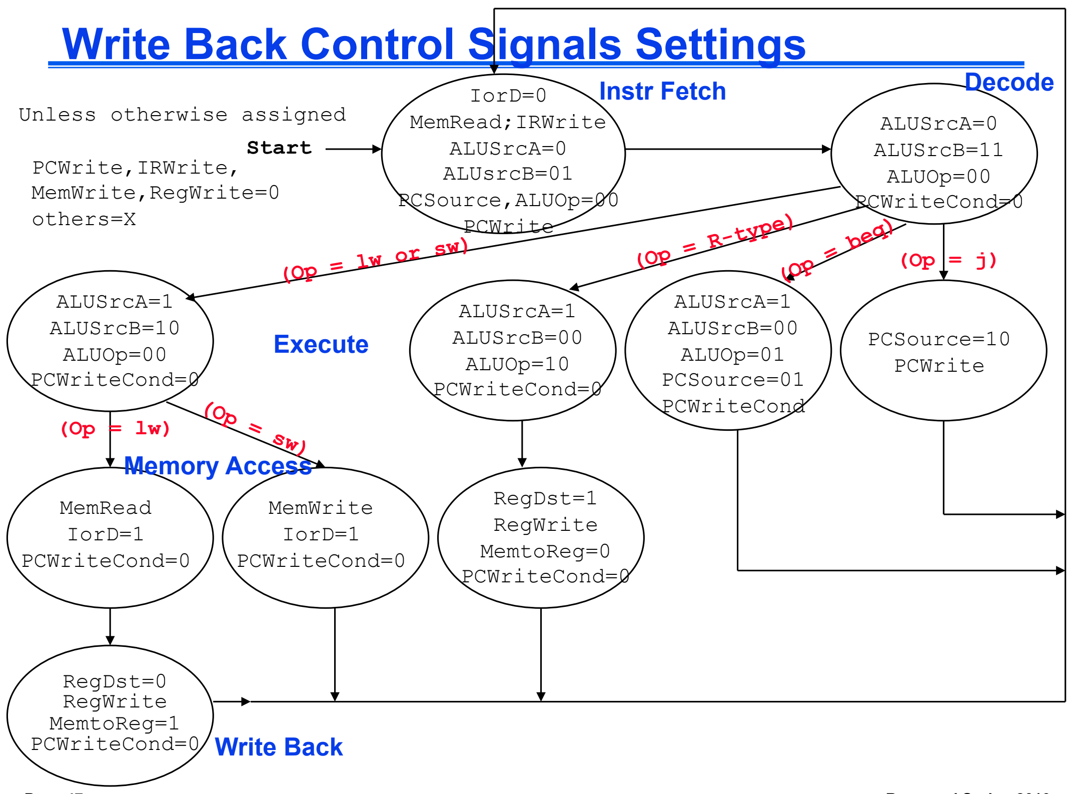
```
Reg[IR[20-16]]= MDR;
```

What about all the other instructions?

Datapath Activity During `lw` Write Back



Write Back Control Signals Settings



RTL Summary

Step	R-type	Mem Ref	Branch	Jump
Instr fetch	IR = Memory[PC]; PC = PC + 4;			
Decode	A = Reg[IR[25-21]]; B = Reg[IR[20-16]]; ALUOut = PC +(sign-extend(IR[15-0])<< 2);			
Execute	ALUOut = A op B;	ALUOut = A + sign-extend (IR[15-0]);	if (A==B) PC = ALUOut;	PC = PC [31-28] ||(IR [25-0] << 2);
Memory access	Reg[IR [15-11]] = ALUOut;	MDR = Memory [ALUOut]; or Memory[ALUOut] = B;		
Write-back		Reg[IR[20-16]] = MDR;		

Answering Simple Questions

- How many cycles will it take to execute this code?

```
        lw    $t2, 0($t3)         5
        lw    $t3, 4($t3)         5
        beq   $t2, $t3, Label     3
   #assume not
        add   $t5, $t2, $t3       4
        sw    $t5, 8($t3)         4
Label:  ...                       =
                                  21 cycles
```

address for second lw being calculated

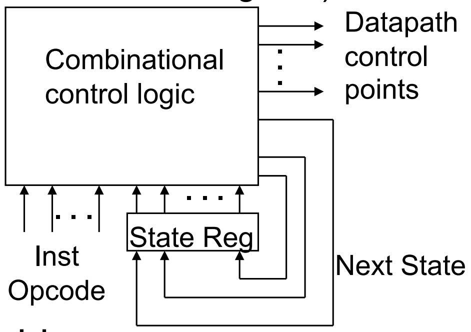
- What is going on during the 8th cycle of execution?
- In what cycle does the actual addition of `$t2` and `$t3` takes place? 16th cycle

12th cycle

- In what cycle is the branch target address calculated?

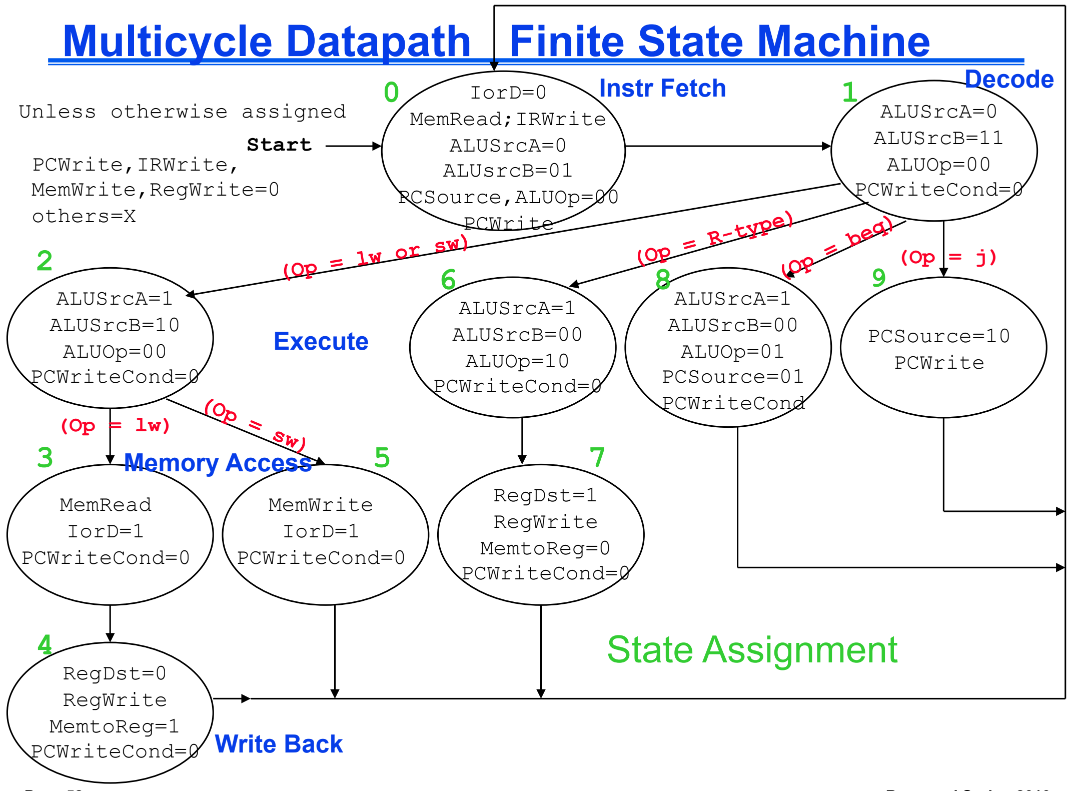
Multicycle Control

- Multicycle datapath control signals are not determined solely by the bits in the instruction
  - e.g., op code bits tell what operation the ALU should be doing, but *not* what instruction cycle is to be done next
- We can use a finite state machine for control
  - a set of states (current state stored in State Register)
  - next state function (determined by current state and the input)
  - output function (determined by current state)

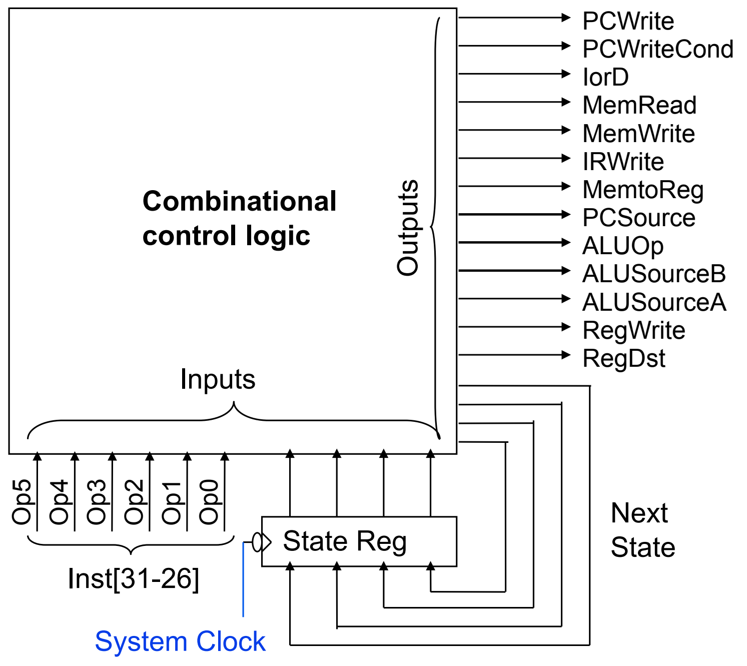


- So we are using a Moore machine (datapath control signals based only on current state)

Multicycle Datapath Finite State Machine



Finite State Machine Implementation



Datapath Control Outputs Truth Table

Outputs	Input Values (Current State[3-0])									
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
PCWrite	1	0	0	0	0	0	0	0	0	1
PCWriteCond	X	0	0	0	0	0	0	0	1	X
IorD	0	X	X	1	X	1	X	X	X	X
MemRead	1	0	0	1	0	0	0	0	0	0
MemWrite	0	0	0	0	0	1	0	0	0	0
IRWrite	1	0	0	0	0	0	0	0	0	0
MemtoReg	X	X	X	X	1	X	X	0	X	X
PCSource	00	XX	XX	XX	XX	XX	XX	XX	01	10
ALUOp	00	00	00	XX	XX	XX	10	XX	01	XX
ALUSrcB	01	11	10	XX	XX	XX	00	XX	00	XX
ALUSrcA	0	0	1	X	X	X	1	X	1	X
RegWrite	0	0	0	0	1	0	0	1	0	0
RegDst	X	X	X	X	0	X	X	1	X	X

Next State Truth Table

Current State [3-0]	Inst[31-26] (Op[5-0])					
	000000 (R-type)	000010 (jmp)	000100 (beq)	100011 (lw)	101011 (sw)	Any other
0000	0001	0001	0001	0001	0001	0001
0001	0110	1001	1000	0010	0010	illegal
0010	XXXX	XXXX	XXXX	0011	0101	illegal
0011	XXXX	XXXX	XXXX	0100	XXXX	illegal
0100	XXXX	XXXX	XXXX	0000	XXXX	illegal
0101	XXXX	XXXX	XXXX	XXXX	0000	illegal
0110	0111	XXXX	XXXX	XXXX	XXXX	illegal
0111	0000	XXXX	XXXX	XXXX	XXXX	illegal
1000	XXXX	XXXX	0000	XXXX	XXXX	illegal
1001	XXXX	0000	XXXX	XXXX	XXXX	illegal

Simplifying the Control Unit Design

- For an implementation of the full MIPS ISA instr's can take from 3 clock cycles to 20+ clock cycles
  - resulting in finite state machines with hundreds to thousands of states with even *more* arcs (state sequences)
    - Such state machine representations become impossibly complex
- Instead, can represent the set of control signals that are asserted during a state as a low-level control "instruction" to be executed by the datapath

microinstructions

- "Executing" the microinstruction is equivalent to asserting the control signals specified by the microinstruction

Microprogramming

- A microinstruction has to specify
  - what control signals should be asserted
  - what microinstruction should be executed next
- Each microinstruction corresponds to one state in the FSM and is assigned a state number (or "address")
  1. Sequential behavior – increment the state (address) of the current microinstruction to get to the state (address) of the next
  2. Jump to the microinstruction that begins execution of the next MIPS instruction (state 0)
  3. Branch to a microinstruction based on control unit input using dispatch tables
     - need one for microinstructions following state 1
     - need another for microinstructions following state 2
- The set of microinstructions that define a MIPS assembly language instruction (macroinstruction) is its microroutine

Defining a Microinstruction Format

- ❑ Format – the fields of the microinstruction and the control signals that are affected by each field
  - ● control signals specified by a field usually have functions that are related
  - ● format is chosen to simplify the representation and to make it difficult to write inconsistent microinstructions
    - i.e., that allow a given control signal be set to two different values
- ❑ Make each field of the microinstruction responsible for specifying a nonoverlapping set of control signals
  - ● signals that are never asserted simultaneously may share the same field
  - ● seven fields for our simple machine
    - ALU control; SRC1; SRC2; Register control; Memory; PCWrite control; Sequencing

Our Microinstruction Format

Field	Value	Signal setting	Comments
ALU control	Add	ALUOp = 00	Cause ALU to add
	Subt	ALUOp = 01	Cause ALU to subtract (compare op for beq)
	Func code	ALUOp = 10	Use IR function code to determine ALU control
SRC1	PC	ALUSrcA = 0	Use PC as top ALU input
	A	ALUSrcA = 1	Use reg A as top ALU input
SRC2	B	ALUSrcB = 00	Use reg B as bottom ALU input
	4	ALUSrcB = 01	Use 4 as bottom ALU input
	Extend	ALUSrcB = 10	Use sign ext output as bottom ALU input
	Extshft	ALUSrcB = 11	Use shift-by-two output as bottom ALU input
Register control	Read		Read RegFile using rs and rt fields of IR as read addr's; put data into A and B
	Write ALU	RegWrite, RegDst = 1, MemtoReg = 0	Write RegFile using rd field of IR as write addr and ALUOut as write data
	Write MDR	RegWrite, RegDst = 0, MemtoReg = 1	Write RegFile using rt field of IR as write addr and MDR as write data

Our Microinstruction Format, con't

Field	Value	Signal setting	Comments
Memory	Read PC	MemRead, IorD = 0,IRWrite	Read memory using PC as addr; write result into IR (and MDR)
	Read ALU	MemRead, IorD = 1	Read memory using ALUOut as addr; write results into MDR
	Write ALU	MemWrite, IorD = 1	Write memory using ALUOut as addr and B as write data
PC write control	ALU	PCSource = 00 PCWrite	Write PC with output of ALU
	ALUOut-cond	PCSource = 01, PCWriteCond	If Zero output of ALU is true, write PC with the contents of ALUOut
	Jump address	PCSource = 10, PCWrite	Write PC with IR jump address after shift-by-two
Sequen-cing	Seq	AddrCtl = 11	Choose next microinstruction sequentially
	Fetch	AddrCtl = 00	Jump to the first microinstruction (i.e., Fetch) to begin a new instruction
	Dispatch 1	AddrCtl = 01	Branch using PLA_1
	Dispatch 2	AddrCtl = 10	Branch using PLA_2

Creating the Microprogram

- Fetch microinstruction

Label (Addr)	ALU control	SRC1	SRC2	Reg control	Memory	PCWrite control	Seq'ing
Fetch (0)	Add	PC	4		Read PC	ALU	Seq

compute PC + 4

fetch instr into IR

write ALU output into PC

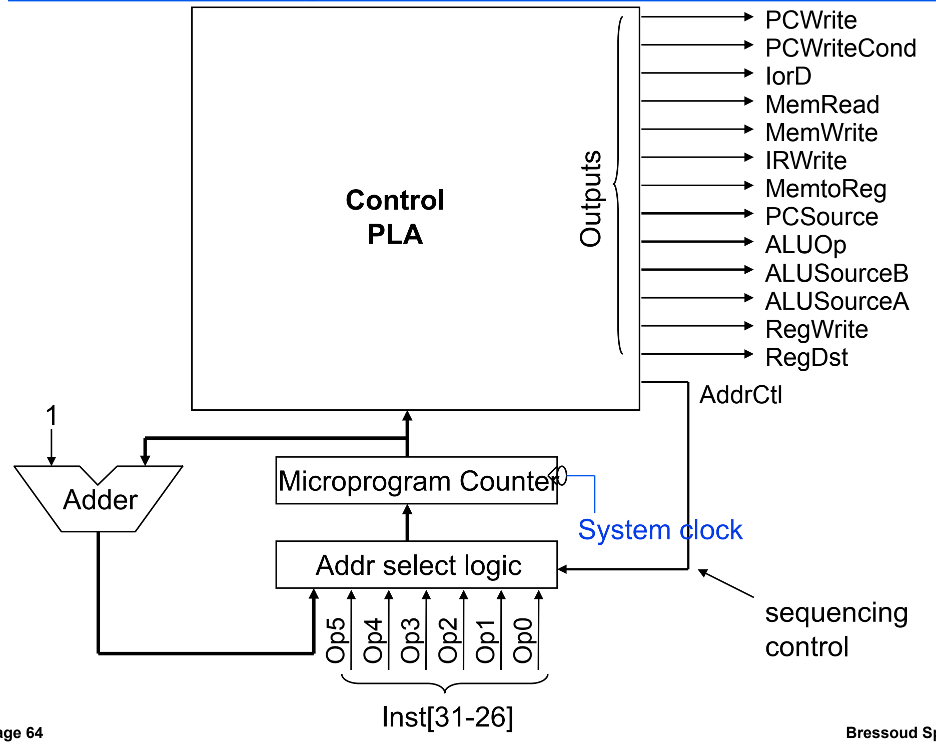
go to μinstr 1

- Label field represents the state (address) of the microinstruction
- Fetch microinstruction assigned state (address) 0

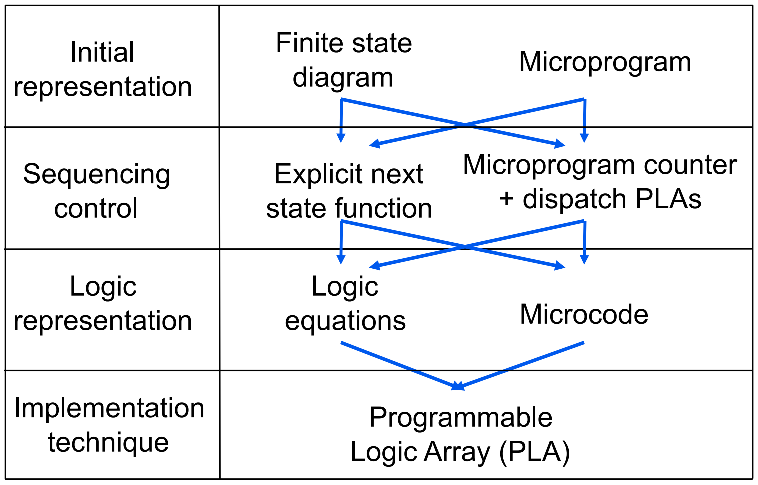
The Entire Control Microprogram

Addr	ALU control	SRC1	SRC2	Reg control	Memory	PCWrite control	Seq'ing
0	Add	PC	4		Read PC	ALU	Seq
1	Add	PC	Ext shft	Read			Disp 1
2	Add	A	Extend				Disp 2
3					Read ALU		Seq
4				Write MDR			Fetch
5					Write ALU		Fetch
6	Func code	A	B				Seq
7				Write ALU			Fetch
8	Subt	A	B			ALUOut-cond	Fetch
9						Jump address	Fetch

Microcode Implementation



Control Path Design Alternatives



Initial representation	Finite state diagram	Microprogram
Sequencing control	Explicit next state function	Microprogram counter + dispatch PLAs
Logic representation	Logic equations	Microcode
Implementation technique	Programmable Logic Array (PLA)	

- ❑ Microprogram representation advantages
  - Easier to design, write, and debug

Multicycle Advantages & Disadvantages

- ❑ Uses the clock cycle efficiently – the clock cycle is timed to accommodate the slowest instruction step
  - balance the amount of work to be done in each step
  - restrict each step to use only one major functional unit
- ❑ Multicycle implementations allow
  - faster clock rates
  - different instructions to take a different number of clock cycles
  - functional units to be used more than once per instruction as long as they are used on different clock cycles

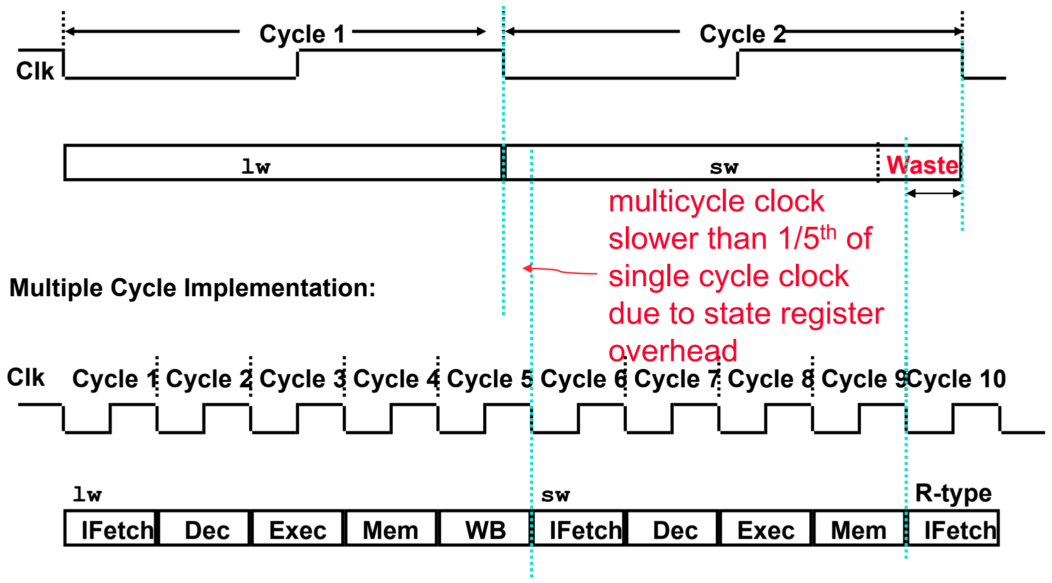
but

- ❑ Requires additional internal state registers, muxes, and more complicated (FSM) control

Single Cycle vs. Multiple Cycle Timing

Single Cycle Implementation:

Multiple Cycle Implementation:


Cycle 1
Cycle 2
Clk
lw
sw
Waste
multicycle clock slower than 1/5th of single cycle clock due to state register overhead
Clk
Cycle 1
Cycle 2
Cycle 3
Cycle 4
Cycle 5
Cycle 6
Cycle 7
Cycle 8
Cycle 9
Cycle 10
lw
sw
R-type
IFetch
Dec
Exec
Mem
WB
IFetch
Dec
Exec
Mem
IFetch