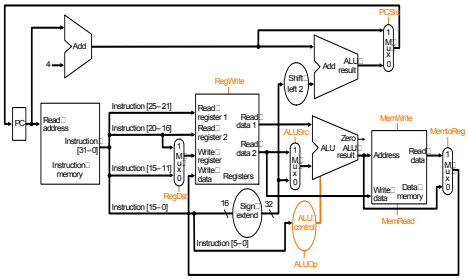
A Complete Datapath for R Type Instructions

- Lw, Sw, Add, Sub, And, Or, Slt can be performed
- For j (jump) we need an additional multiplexor



What Else is Needed in Data Path

- Support for j and jr
  - For both of them PC value need to come from somewhere else
  - For J, PC is created by 4 bits (31:28) from old PC, 26 bits from IR (27:2) and 2 bits are zero (1:0)
  - For JR, PC value comes from a register
- Support for JAL
  - Address is same as for J inst
  - OLD PC needs to be saved in register 31
- And what about immediate operand instructions
  - Second operand from instruction, but without shifting
- Support for other instructions like lw and immediate inst write

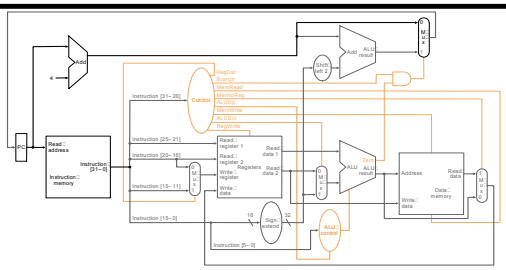
Control

- For each instruction
  - Select the registers to be read (always read two)
  - Select the 2nd ALU input
  - Select the operation to be performed by ALU
  - Select if data memory is to be read or written
  - Select what is written and where in the register file
  - Select what goes in PC
- Information comes from the 32 bits of the instruction
- Example:

add $8, $17, $18 Instruction Format:

000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct

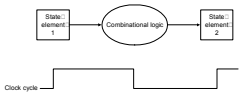
Adding Control to DataPath



Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Our Simple Control Structure

- All of the logic is combinational
- We wait for everything to settle down, and the right thing to be done
  - ALU might not produce “right answer” right away
  - we use write signals along with clock to determine when to write
- Cycle time determined by length of the longest path



We are ignoring some details like setup and hold times

ALU Control

- ALU's operation based on instruction type and function code
  - e.g., what should the ALU do with any instruction
- Example: lw $1, 100($2)
-

35	2	1	100
op	rs	rt	16 bit offset

- ALU control input

```
000   AND
001   OR
010   add
110   subtract
111   set-on-less-than
```

- Why is the code for subtract 110 and not 011?

Other Control Information

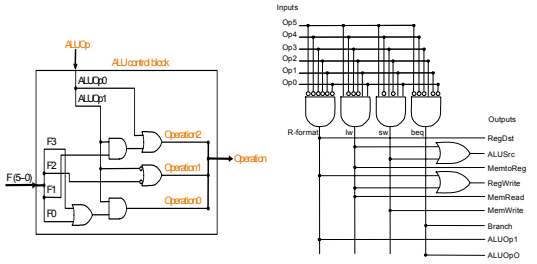
- Must describe hardware to compute 3-bit ALU conrol input
  - given instruction type
    - 00 = lw, sw
    - 01 = beq,
    - 10 = arithmetic
    - 11 = Jump

    ALUOp computed from instruction type
  - function code for arithmetic
- Control can be described using a truth table:

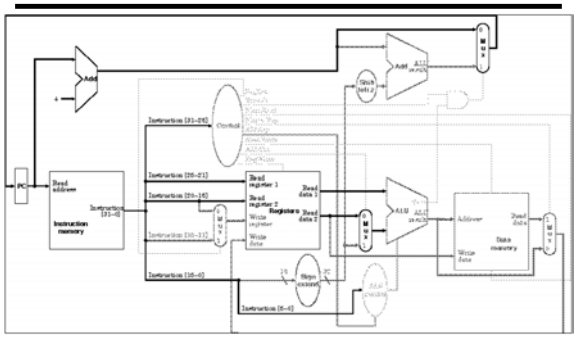
ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

Implementation of Control

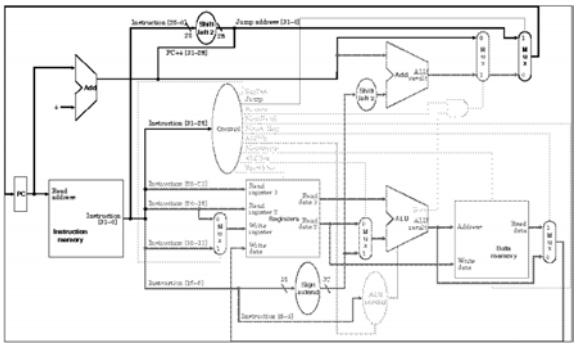
- Simple combinational logic to realize the truth tables



A Complete Datapath with Control

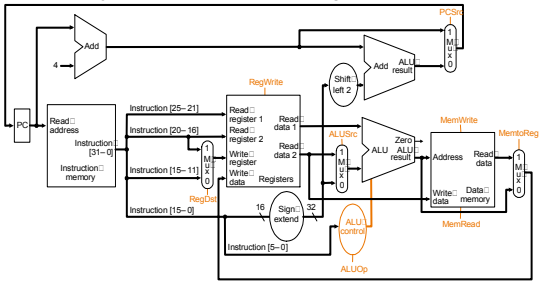


Datapath with Control and Jump Instruction



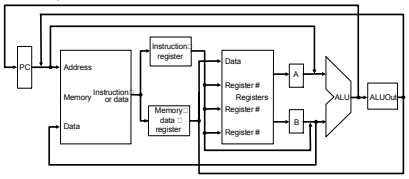
Timing: Single Cycle Implementation

- Calculate cycle time assuming negligible delays except:
  - memory (2ns), ALU and adders (2ns), register file access (1ns)

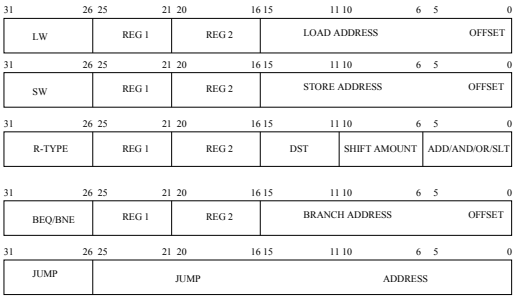


Where we are headed

- Single Cycle Problems:
  - what if we had a more complicated instruction like floating point?
  - wasteful of area
- One Solution:
  - use a "smaller" cycle time
  - have different instructions take different numbers of cycles
  - a "multicycle" datapath:



Instruction Format



Operation for Each Instruction

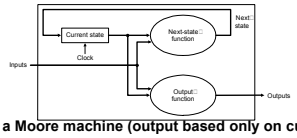
LW:	SW:	R-Type:	BR-Type:	JMP-Type:
1. READ INST	1. READ INST	1. READ INST	1. READ INST	1. READ INST
2. READ REG 1 *READ REG 2*	2. READ REG 1 READ REG 2	2. READ REG 1 READ REG 2	2. READ REG 1 READ REG 2	2.
3. ADD REG 1 + OFFSET	3. ADD REG 1 + OFFSET	3. OPERATE on REG 1 / REG 2	3. SUB REG 2 from REG 1	3.
4. READ MEM	4. WRITE MEM	4.	4.	4.
5. WRITE REG2	5.	5. WRITE DST	5.	5.

Multicycle Approach

- We will be reusing functional units
  - Break up the instruction execution in smaller steps
  - Each functional unit is used for a specific purpose in one cycle
  - Balance the work load
  - ALU used to compute address and to increment PC
  - Memory used for instruction and data
- At the end of cycle, store results to be used again
  - Need additional registers
- Our control signals will not be determined solely by instruction
  - e.g., what should the ALU do for a "subtract" instruction?
- We'll use a finite state machine for control

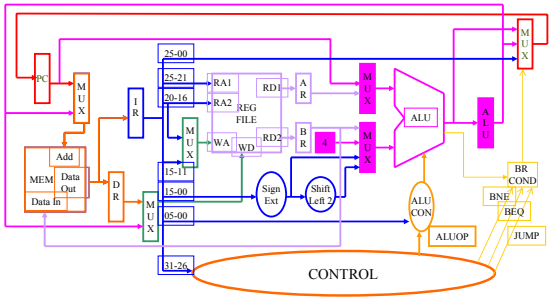
Review: finite state machines

- Finite state machines:
  - a set of states and
  - next state function (determined by current state and the input)
  - output function (determined by current state and possibly input)



  - We'll use a Moore machine (output based only on current state)

Multi- Cycle DataPath Operation



Five Execution Steps

- Instruction Fetch
- Instruction Decode and Register Fetch
- Execution, Memory Address Computation, or Branch Completion
- Memory Access or R-type instruction completion
- Write-back step

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

Step 1: Instruction Fetch

- Use PC to get instruction 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 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?

Step 2: Instruction Decode and Register Fetch

- Read registers rs and rt in case we need them
- Compute the branch address in case the instruction is a branch
- RTL:

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

- We aren't setting any control lines based on the instruction type
  (we are busy "decoding" it in our control logic)

Step 3 (instruction dependent)

- ALU is performing one of three functions, based on instruction type
- Memory Reference:

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

- R-type:

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

- Branch:

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


Step 4 (R type or memory access)

- Loads and stores access memory

```
MDR = Memory[ALUOut];
      or
Memory[ALUOut] = B;
```

- R-type instructions finish

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

The write actually takes place at the end of the cycle on the edge

Write back step

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

What about all the other instructions?

Summary:

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR = Memory[PC] PC = PC + 4			
Instruction decode/register fetch	A = Reg [IR[25-21]] B = Reg [IR[20-16]] ALUOut = PC + (sign-extend (IR[15-0]) << 2)			
Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend (IR[15-0])	if (A ==B) then PC = ALUOut	PC = PC [31-28] II (IR[25-0]<<2)
Memory access or R-type completion	Reg [IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
Memory read completion		Load: Reg[IR[20-16]] = MDR		