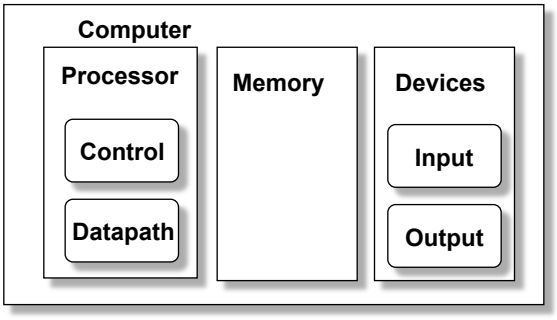
0111
A *Single Cycle CPU*

ENGR 3410 – Computer Architecture
Fall 2010

Datapath & Control

- Readings 4.1 – 4.4



- Datapath: System for performing operations on data, plus memory access.
- Control: Control the datapath in response to instructions.

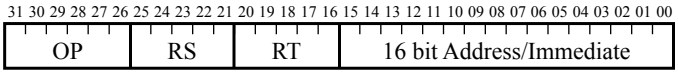
Simple CPU

Develop complete CPU for subset of instruction set

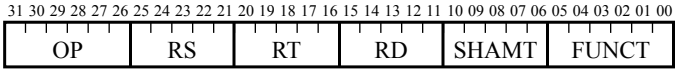
Memory: `lw, sw`

Branch: `beq`

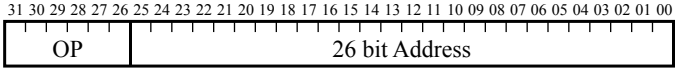
Arithmetic: `addi`



Arithmetic: `add, sub`

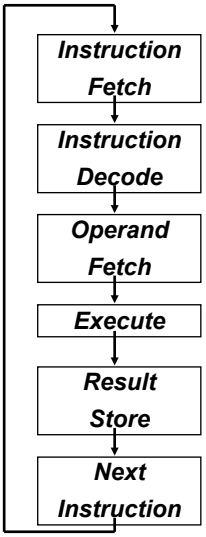


Jump: `j`



Most other instructions similar

Execution Cycle

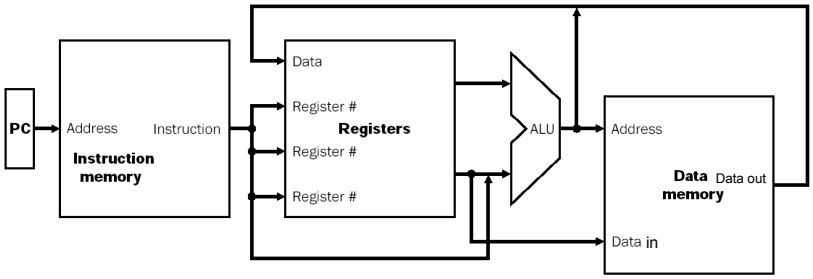


- **Obtain instruction from program storage**
- **Determine required actions and instruction size**
- **Locate and obtain operand data**
- **Compute result value or status**
- **Deposit results in storage for later use**
- **Determine successor instruction**

Processor Overview

Overall Dataflow

- PC fetches instructions
- Instructions select operand registers, ALU immediate values
- ALU computes values
- Load/Store addresses computed in ALU
- Result goes to register file or Data memory



Processor Design

Convert instructions to Register Transfer Level (RTL) specification

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

RTL specifies required interconnection of units and specifies semantics

Control designed to achieve given paths for each instruction

Instruction Fetch

```
Instruction = Mem[PC];      // Fetch Instruction
PC = PC + 4;                // Increment PC (32bit)
```

Instruction Fetch

```
Instruction = Mem[PC];      // Fetch Instruction
PC = PC + 4;                // Increment PC (32bit)
```

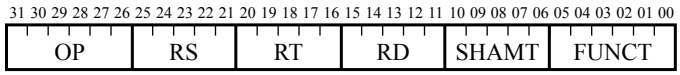
Add/Subtract RTL

Add instruction: `add rd, rs, rt`

```
Instruction = Mem[PC];
Reg[rd] = Reg[rs] + Reg[rt];
PC = PC + 4;
```

Subtract instruction: `sub rd, rs, rt`

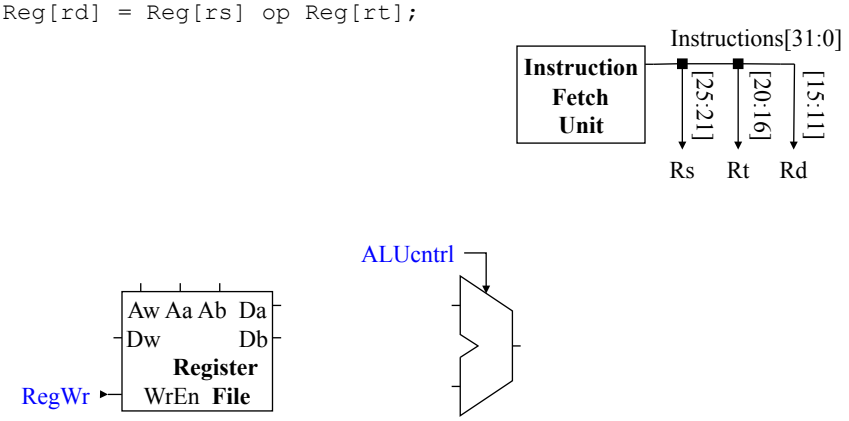
```
Instruction = Mem[PC];
Reg[rd] = Reg[rs] - Reg[rt];
PC = PC + 4;
```



R-format

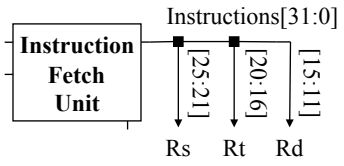
Datapath for Reg/Reg Ops

```
Reg[rd] = Reg[rs] op Reg[rt];
```

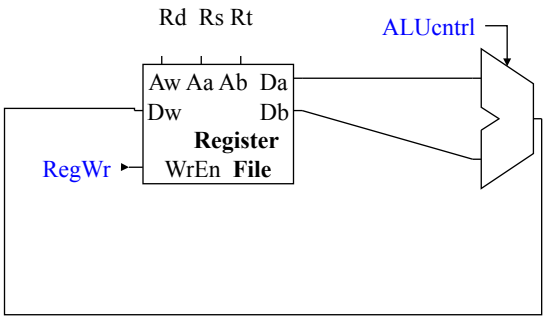


Datapath for Reg/Reg Ops

```
Reg[rd] = Reg[rs] op Reg[rt];
```



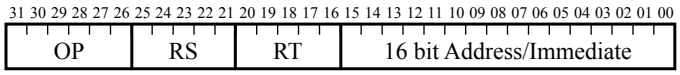
ALUcntrl based on Function, OP
RegWr true when doing such an OP



Add Immediate RTL

Add immediate instruction: `addi rt, rs, imm`

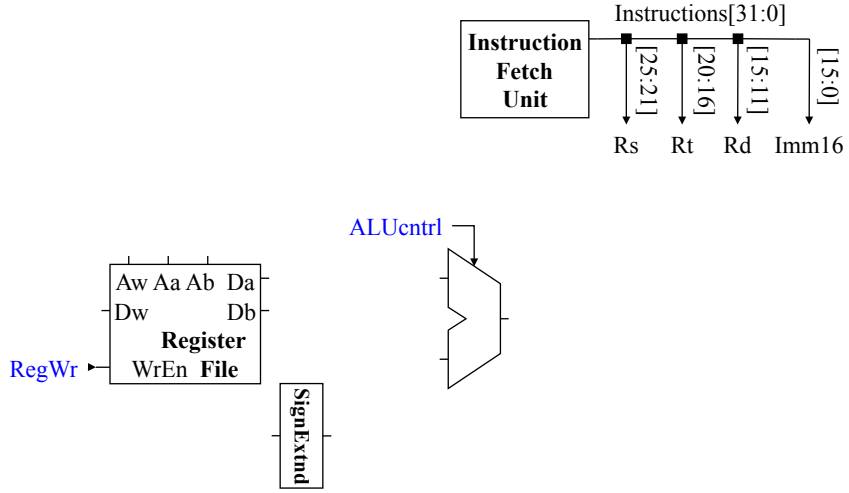
```
Instruction = Mem[PC];
Reg[rt] = Reg[rs] + SignExtend(imm);
PC = PC + 4;
```



I-format

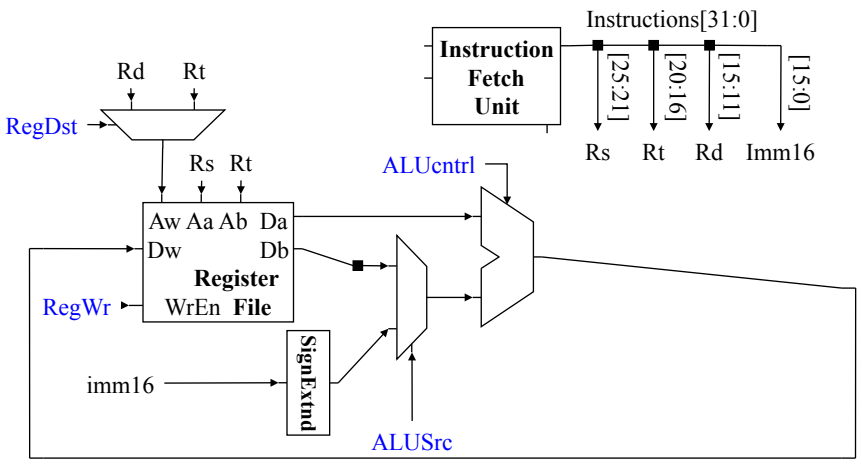
Datapath + Immediate Ops

```
Reg[rt] = Reg[rs] + SignExtend(imm);
```



Datapath + Immediate Ops

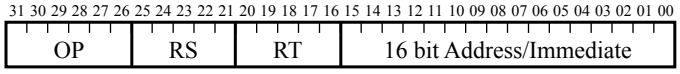
```
Reg[rt] = Reg[rs] + SignExtend(imm);
```



Load RTL

Load Instruction: `lw rt, imm(rs)`

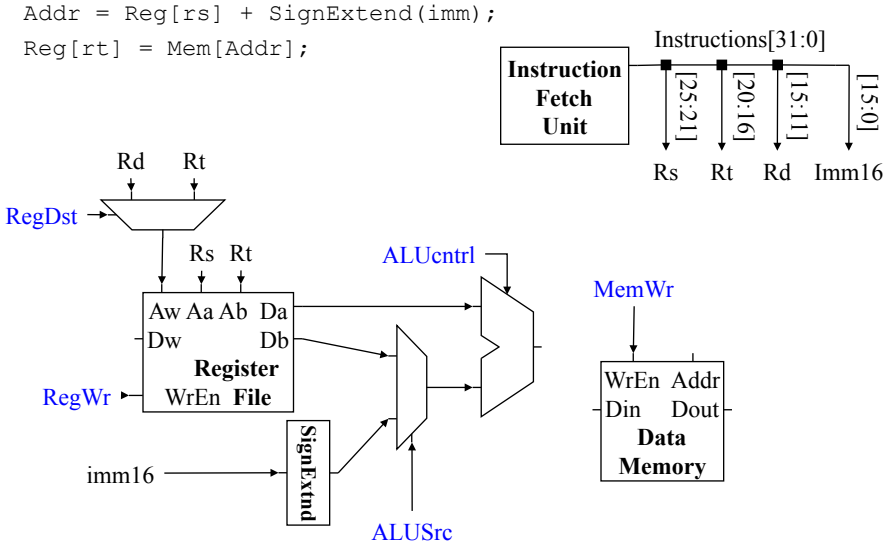
```
Instruction = Mem[PC];
Addr = Reg[rs] + SignExtend(imm);
Reg[rt] = Mem[Addr];
PC = PC + 4;
```



I-format

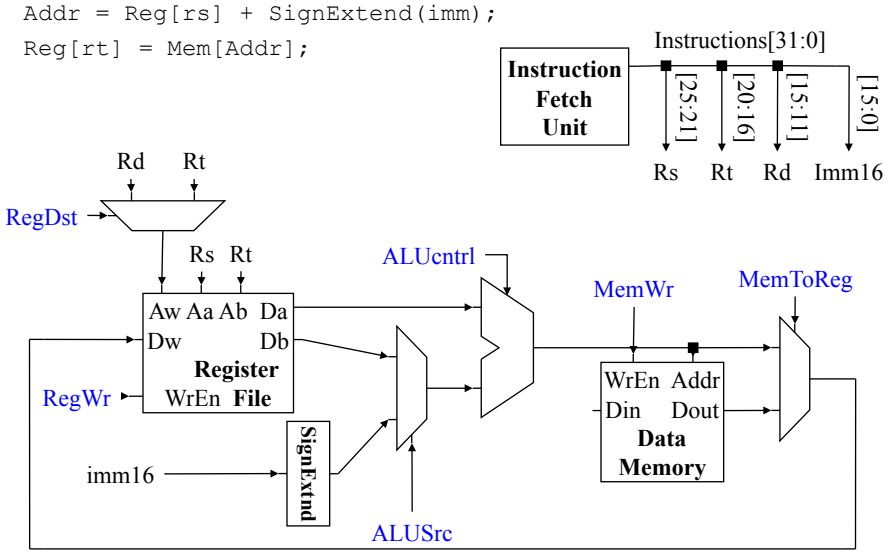
Datapath + Load

```
Addr = Reg[rs] + SignExtend(imm);
Reg[rt] = Mem[Addr];
```



Datapath + Load

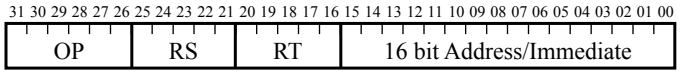
```
Addr = Reg[rs] + SignExtend(imm);
Reg[rt] = Mem[Addr];
```



Store RTL

Store Instruction: `sw rt, imm(rs)`

```
Instruction = Mem[PC];
Addr = Reg[rs] + SignExtend(imm);
Mem[Addr] = Reg[rt];
PC = PC + 4;
```

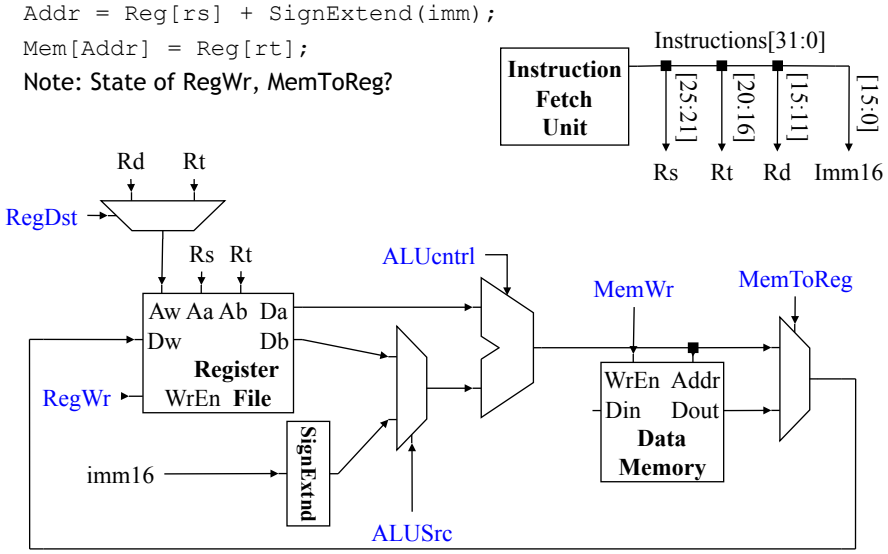


I-format

Datapath + Store

```
Addr = Reg[rs] + SignExtend(imm);
Mem[Addr] = Reg[rt];
```

Note: State of RegWr, MemToReg?

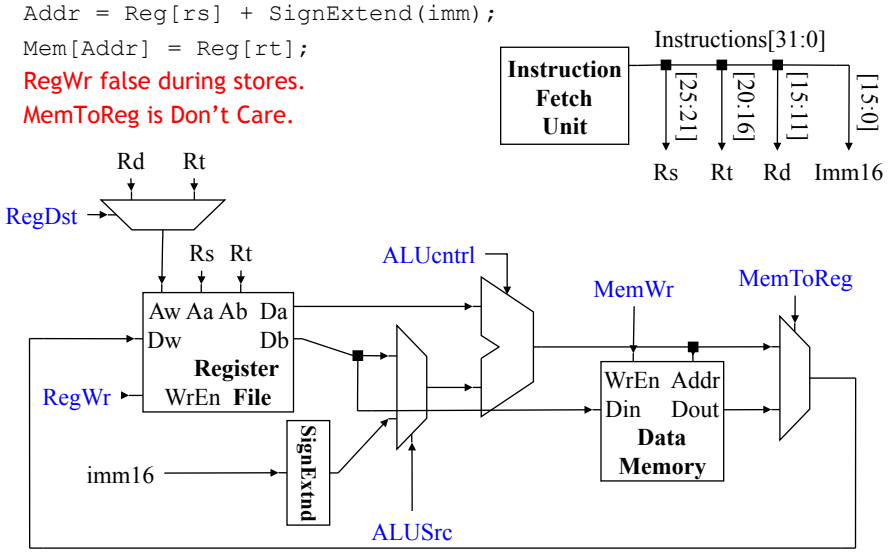


Datapath + Store

```
Addr = Reg[rs] + SignExtend(imm);
Mem[Addr] = Reg[rt];
```

RegWr false during stores.

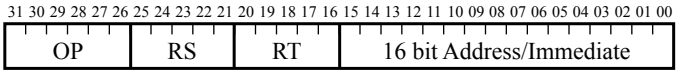
MemToReg is Don't Care.



Branch RTL

Branch Instruction: `beq rs, rt, imm`

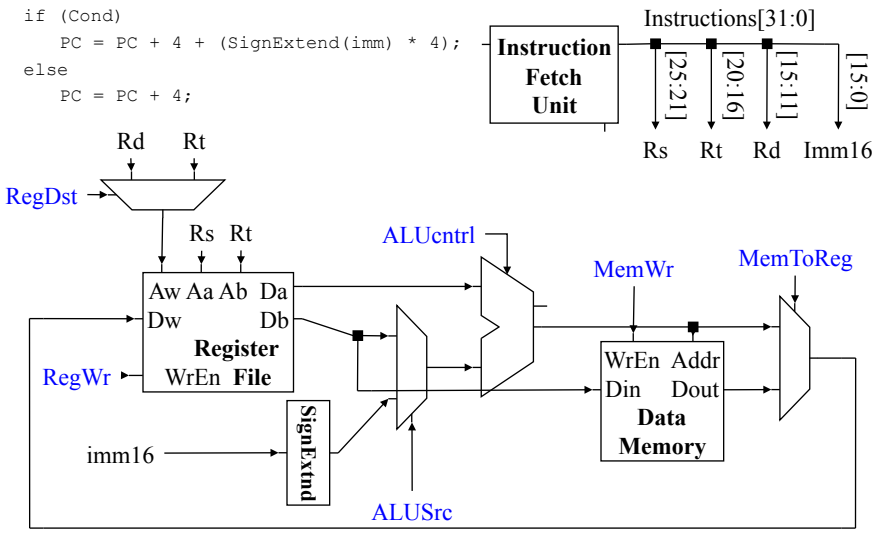
```
Instruction = Mem[PC];
Cond = (Reg[rs] - Reg[rt]) == 0;          // Test equality
if (Cond)
     PC = PC + 4 + (SignExtend(imm) * 4);  // Neg for backward
                                           // *4: LSbits == 00
else
     PC = PC + 4;
```



I-format

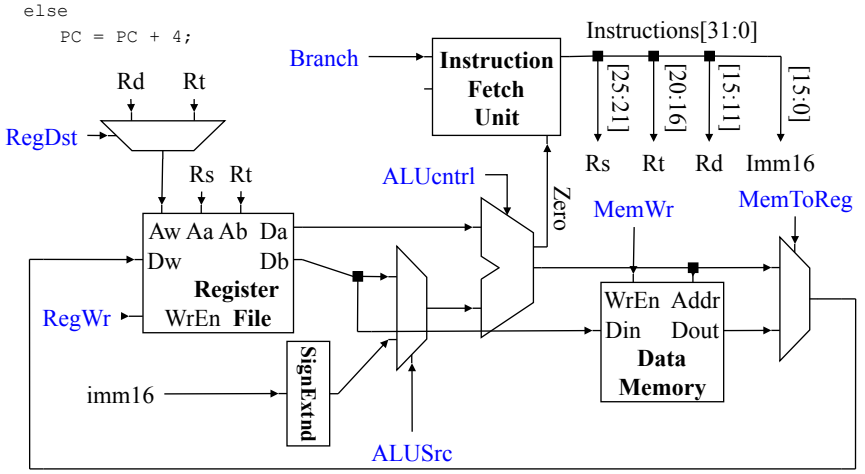
Datapath + Branch

```
Cond = (Reg[rs] - Reg[rt]) == 0;
if (Cond)
   PC = PC + 4 + (SignExtend(imm) * 4);
else
   PC = PC + 4;
```



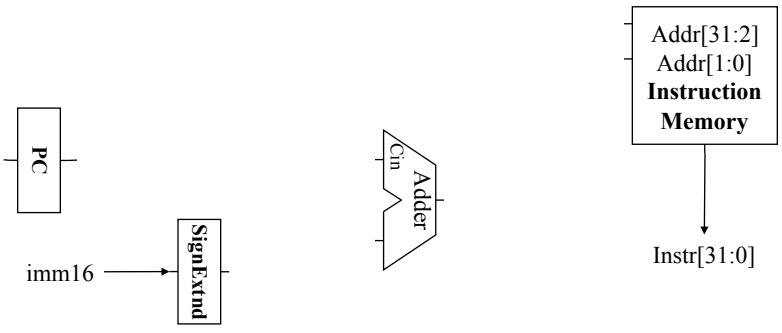
Datapath + Branch

```
Cond = (Reg[rs] - Reg[rt]) == 0;
if (Cond)
    PC = PC + 4 + (SignExtend(imm) * 4);
else
    PC = PC + 4;
```



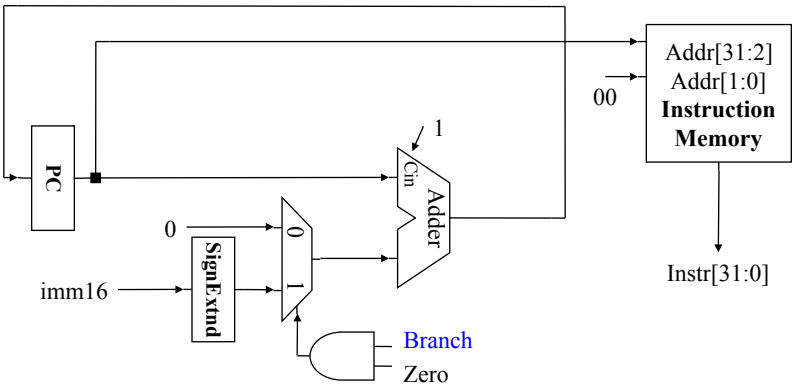
Instruction Fetch + Branch

```
Cond = (Reg[rs] - Reg[rt]) == 0;
if (Cond)
    PC = PC + 4 + SignExtend(imm)*4;
else
    PC = PC + 4;
```



Instruction Fetch + Branch

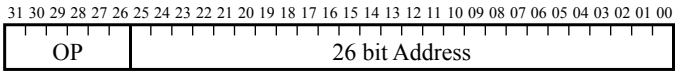
```
Cond = (Reg[rs] - Reg[rt]) == 0;
if (Cond)
   PC = PC + 4 + (SignExtend(imm) * 4);
else
   PC = PC + 4;
```



Jump RTL

Store Instruction: `j target`

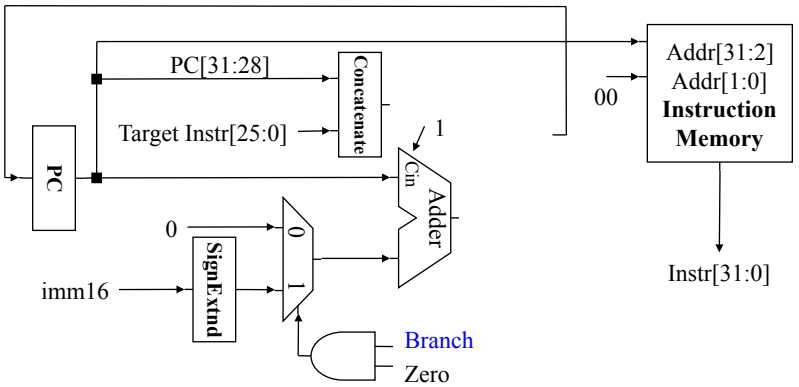
```
Instruction = Mem[PC];
PC = { PC[31:28], target[25:0], "00" };
```



J-format

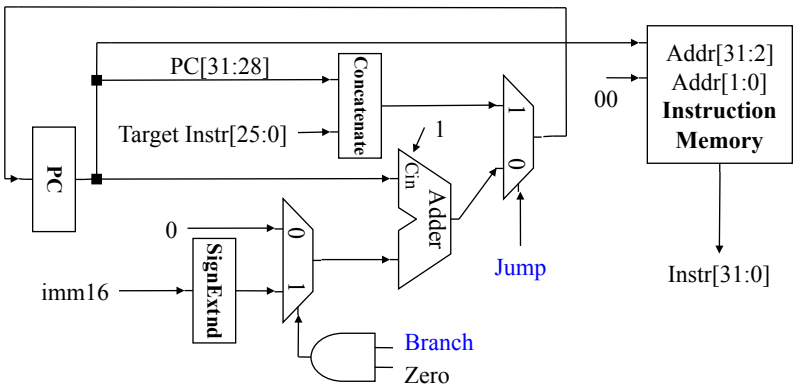
Instruction Fetch + Jump

```
PC = { PC[31:28], target[25:0], "00" };
```

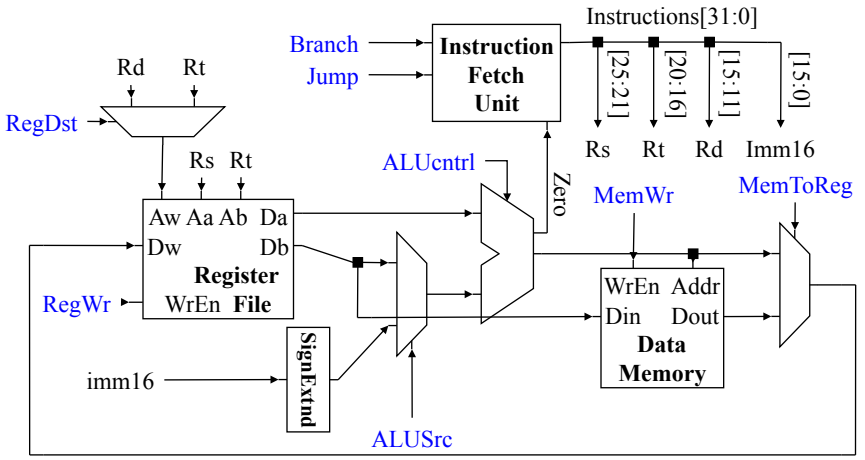


Instruction Fetch + Jump

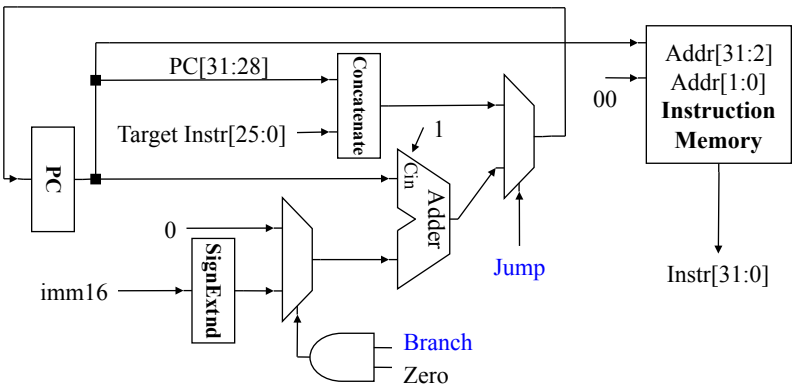
```
PC = { PC[31:28], target[25:0], "00" };
```



Complete Datapath



Complete Fetch Unit



Control

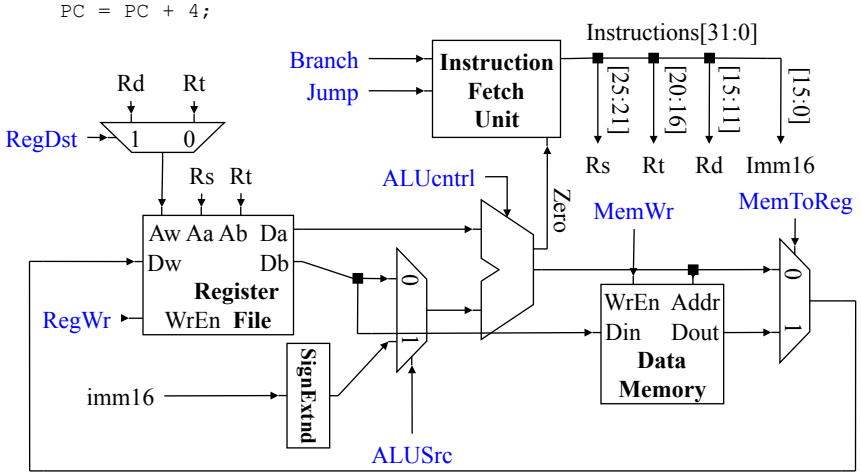
- Identify control points for pieces of datapath
  - Instruction Fetch Unit
  - ALU
  - Memories
  - Datapath muxes
  - Etc.
- Use RTL for determine per-instruction control assignments

Control Signals

Func	100000	100010	XXXXXX	XXXXXX	XXXXXX	XXXXXX
Op	000000	000000	100011	101011	000100	000010
	add	sub	lw	sw	beq	j

Add Control

```
add rd, rs, rt
   Instruction = Mem[PC];
   Reg[rd] = Reg[rs] + Reg[rt];
   PC = PC + 4;
```

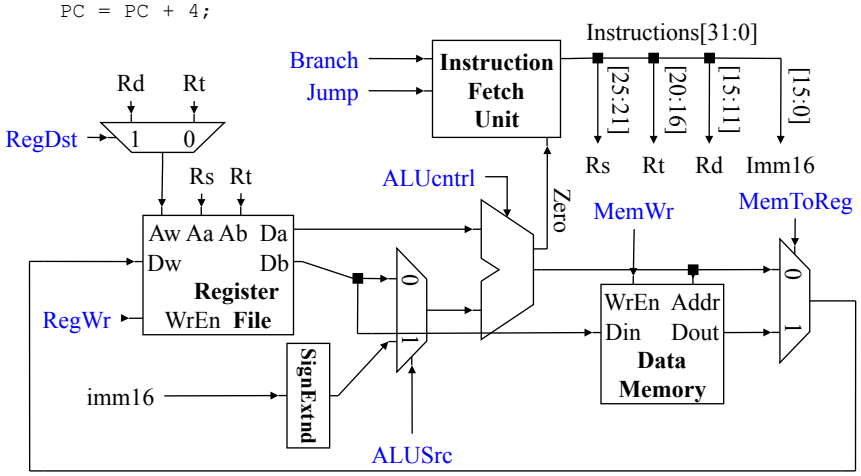


Control Signals

Func	100000	100010	XXXXXX	XXXXXX	XXXXXX	XXXXXX
Op	000000	000000	100011	101011	000100	000010
	add	sub	lw	sw	beq	j
RegDst	1					
ALUSrc	0					
MemToReg	0					
RegWr	1					
MemWr	0					
Branch	0					
Jump	0					
ALUCntrl	Add					

Subtract Control

```
sub rd, rs, rt
   Instruction = Mem[PC];
   Reg[rd] = Reg[rs] - Reg[rt];
   PC = PC + 4;
```

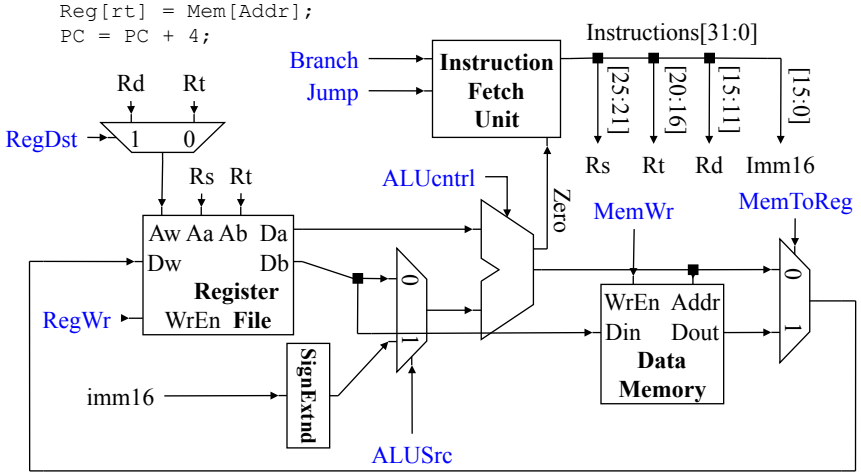


Control Signals

Func	100000	100010	XXXXXX	XXXXXX	XXXXXX	XXXXXX
Op	000000	000000	100011	101011	000100	000010
	add	sub	lw	sw	beq	j
RegDst	1	1				
ALUSrc	0	0				
MemToReg	0	0				
RegWr	1	1				
MemWr	0	0				
Branch	0	0				
Jump	0	0				
ALUCntrl	Add	Sub				

Load Control

```
lw rt, imm(rs)
   Instruction = Mem[PC];
   Addr = Reg[rs] + SignExtend(imm);
   Reg[rt] = Mem[Addr];
   PC = PC + 4;
```

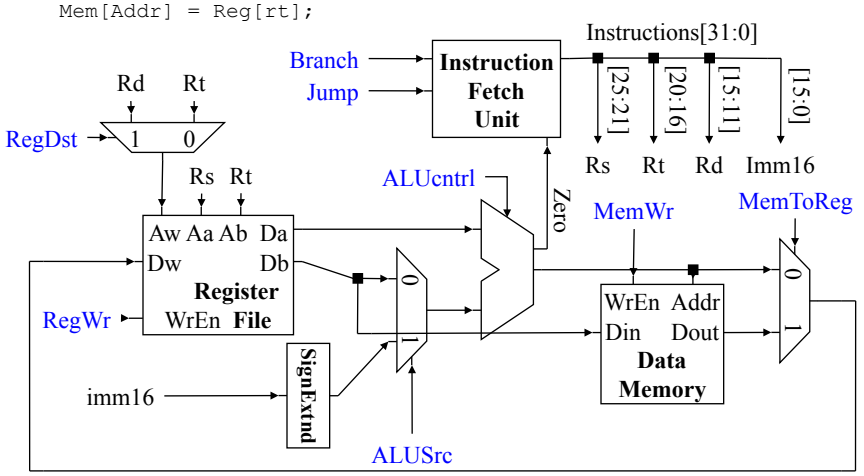


Control Signals

Func	100000	100010	XXXXXX	XXXXXX	XXXXXX	XXXXXX
Op	000000	000000	100011	101011	000100	000010
	add	sub	lw	sw	beq	j
RegDst	1	1	0			
ALUSrc	0	0	1			
MemToReg	0	0	1			
RegWr	1	1	1			
MemWr	0	0	0			
Branch	0	0	0			
Jump	0	0	0			
ALUCntrl	Add	Sub	Add			

Store Control

```
sw rt, imm(rs)
   Instruction = Mem[PC];
   Addr = Reg[rs] + SignExtend(imm);
   Mem[Addr] = Reg[rt];
```

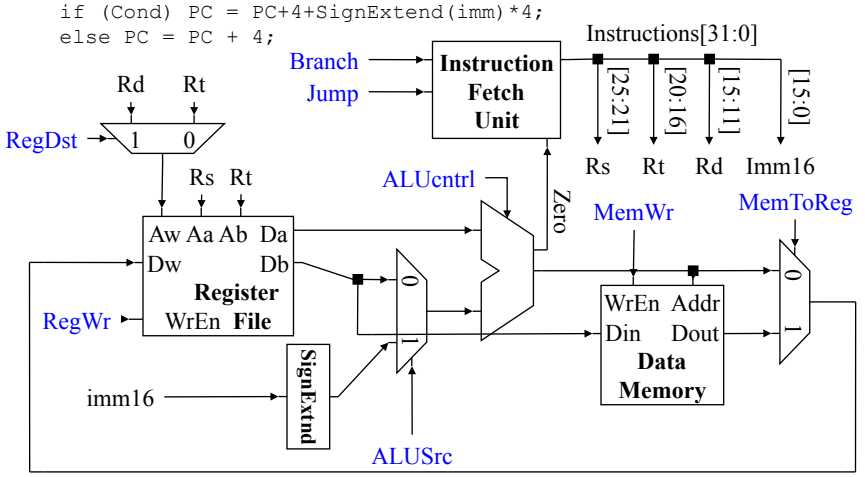


Control Signals

Func	100000	100010	XXXXXX	XXXXXX	XXXXXX	XXXXXX
Op	000000	000000	100011	101011	000100	000010
	add	sub	lw	sw	beq	j
RegDst	1	1	0	X		
ALUSrc	0	0	1	1		
MemToReg	0	0	1	X		
RegWr	1	1	1	0		
MemWr	0	0	0	1		
Branch	0	0	0	0		
Jump	0	0	0	0		
ALUCntrl	Add	Sub	Add	Add		

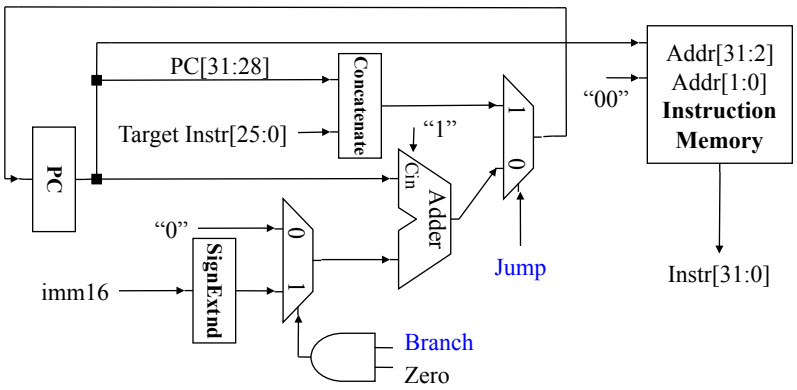
Branch Control

```
beq rs, rt, imm
   Instruction = Mem[PC];
   Cond = (Reg[rs] - Reg[rt]) == 0;
   if (Cond) PC = PC+4+SignExtend(imm)*4;
   else PC = PC + 4;
```



Branch Control (cont.)

```
beq rs, rt, imm
   Instruction = Mem[PC];
   Cond = (Reg[rs] - Reg[rt]) == 0;
   if (Cond) PC = PC+4+SignExtend(imm)*4;
   else PC = PC + 4;
```

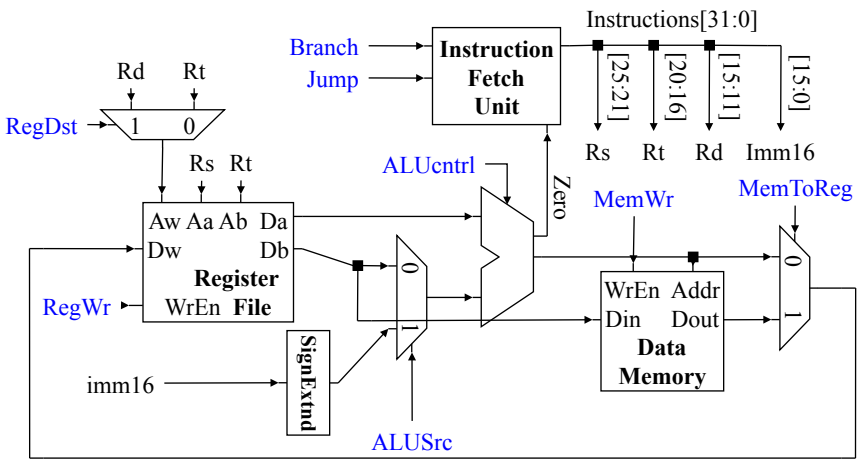


Control Signals

Func	100000	100010	XXXXXX	XXXXXX	XXXXXX	XXXXXX
Op	000000	000000	100011	101011	000100	000010
	add	sub	lw	sw	beq	
RegDst	1	1	0	X	X	
ALUSrc	0	0	1	1	0	
MemToReg	0	0	1	X	X	
RegWr	1	1	1	0	0	
MemWr	0	0	0	1	0	
Branch	0	0	0	0	1	
Jump	0	0	0	0	0	
ALUCntrl	Add	Sub	Add	Add	Sub	

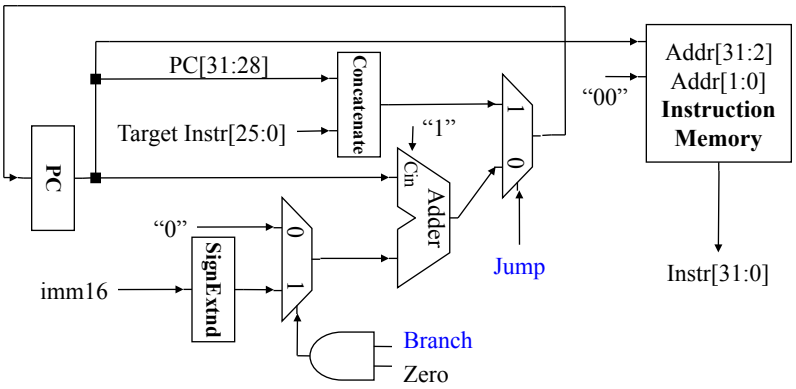
Jump Control

```
j target
   Instruction = Mem[PC];
   PC = { PC[31:28], target[25:0], "00" };
```



Jump Control (cont.)

```
j target
   Instruction = Mem[PC];
   PC = { PC[31:28], target[25:0], "00" };
```



Control Signals

Func	100000	100010	XXXXXX	XXXXXX	XXXXXX	XXXXXX
Op	000000	000000	100011	101011	000100	000010
	add	sub	lw	sw	beq	j
RegDst	1	1	0	X	X	X
ALUSrc	0	0	1	1	0	X
MemToReg	0	0	1	X	X	X
RegWr	1	1	1	0	0	0
MemWr	0	0	0	1	0	0
Branch	0	0	0	0	1	X
Jump	0	0	0	0	0	1
ALUCntrl	Add	Sub	Add	Add	Sub	X