

## Announcements

- Upcoming deadlines
  - Lab1 is due today
    - Demo by 5pm, report by midnight
  - HW2 due on Thursday 2/1
  - PA1 due on Thursday 2/8
- Lab #2 is out
  - Learn from previous lab: start early
- Quiz #1: Tue 2/6, 7pm–9pm (location TBD)
  - Catch-up with reading material

## Lecture 7

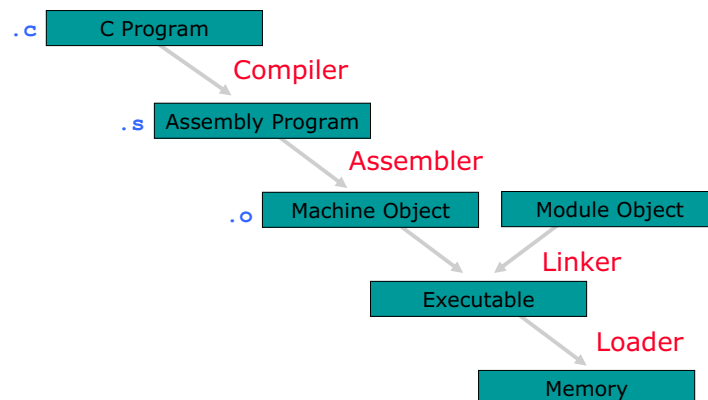
### Building A Simple Processor

Christos Kozyrakis  
Stanford University

<http://eeclass.stanford.edu/ee108b>

## Review: Translation Hierarchy

- High-level → Assembly → Machine



## Review: Code Optimization

- Goal: Improve performance by:
  - Removing redundant work
    - Unreachable code
    - Common-subexpression elimination
    - Induction variable elimination
  - Creating simpler operations
    - Dealing with constants in the compiler
    - Strength reduction
  - Managing registers well

$$\text{Execution Time} = \text{Instructions} \times \text{CPI} \times \text{Clock Cycle Time}$$

## Assembler

- Expands macros and pseudoinstructions as well as converts constants
- Primary purpose is to produce an object file
  - Machine language instructions
  - Application data
  - Information for memory organization

## Object File

- Includes

Object header	Describes file organization
Text segment	Machine code
Data segment	Static data (initialized)
Relocation information	Identifies instruction & data that depend on absolute address when the program is loaded
Symbol table	List of labels that are not defined (e.g. external references)
Debugging information	Describes relationship between source code and machine instructions

## Linker

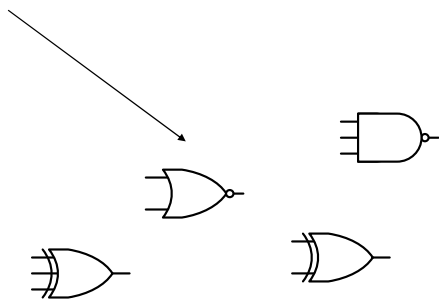
- Linker combines multiple object modules
  - Identify where code/data will be placed in memory
  - Resolve code/data cross references
  - Produces executable if all references found
- Steps
  1. Place code and data modules in memory
  2. Determine the address of data and instruction labels
  3. Patch both the internal and external references
- Separation between compiler and linker makes standard libraries an efficient solution to maintaining modular code

## Loader

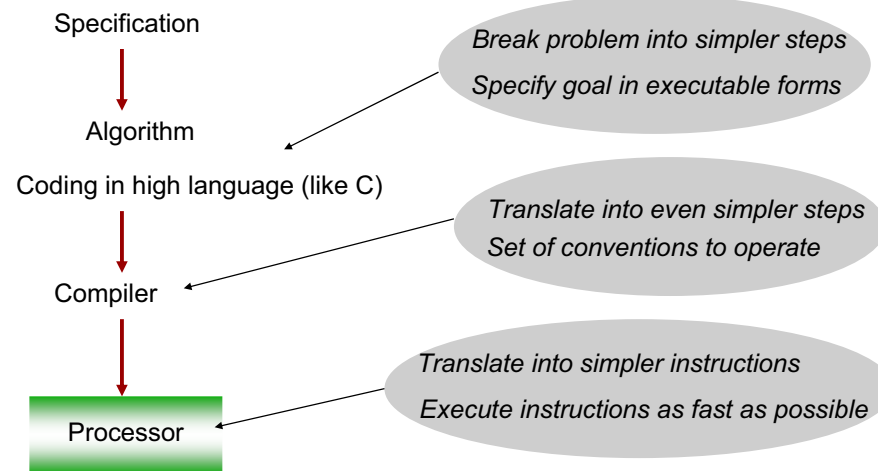
- Loader used at run-time
  1. Reads executable file header for size of text/data segments
  2. Create address space sufficiently large
  3. Copy program from executable on disk into memory
  4. Copy arguments to main program's stack
  5. Initialize machine registers and set stack pointer
  6. Jump to start-up routine
  7. Terminate program when execution completes

## What We Really Want?

- To go from
  - Some goal like **Find N!**



## How to solve a complex problem



## How To Build A Processor (or any complex hardware)

- Break operation down to steps even gates can understand
- Generally decompose task into two kinds of operations
  - Things that deal with the real data (Datapath)
  - Things that control the stuff operating on the real data (Control)
- Find a decomposition that is simple, and efficient
  - Some are obvious, others can be more subtle
- We will start simple
  - Add stuff to improve performance

## How to Execute Instructions

- First we need to:
- Then we need to:
- Then we need to:
- Then we need to:
- Finally we need to:

## How to Execute Instructions

- First we need to:
  - Fetch the instruction
- Then we need to:
  - Decode instruction / fetch register
- Then we need to:
  - Do the operation
- Then we need to:
  - Write the result into register-file
- Finally we need to:
  - Calculate the next instruction address

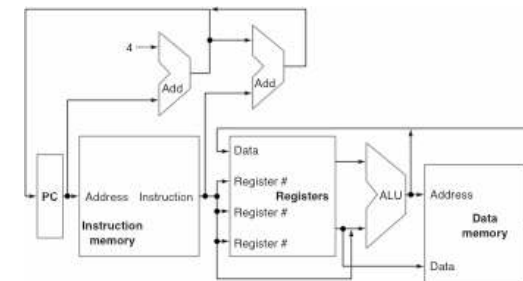
## Subset of Instructions

- To simplify our study of processor design, we will focus on a subset of the MIPS instructions
  - Memory: `lw` and `sw`
  - Arithmetic: `addu`, `subu`, `and`, `ori`, and `slt`
  - Branch: `beq` and `j`
- The method of implementing other instructions should come naturally from these

## Starting The Design

- Think of the steps needed for each instruction execution
  - Become clear quickly we need to create a sequential process
  - Instructions could take multiple cycles or one cycle
- Steps that occur on each instruction:
  - Fetch instruction from memory - address is specified by PC
  - Read one or two registers
  - Do add/sub/etc. using ALU (see Appendix B.5)
  - Fetch a value from memory
  - Store results to register-file/memory
- Needed state for single cycle machine:
  - Instruction pointer (PC), 32 registers, (memory values)

## Starting Dataflow

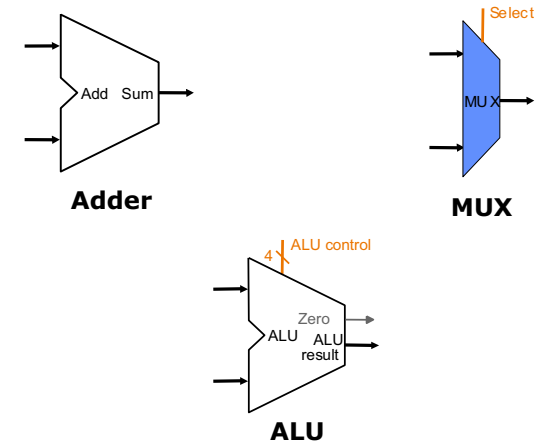


- Major functional units
- Major connections

## Logic Design Review

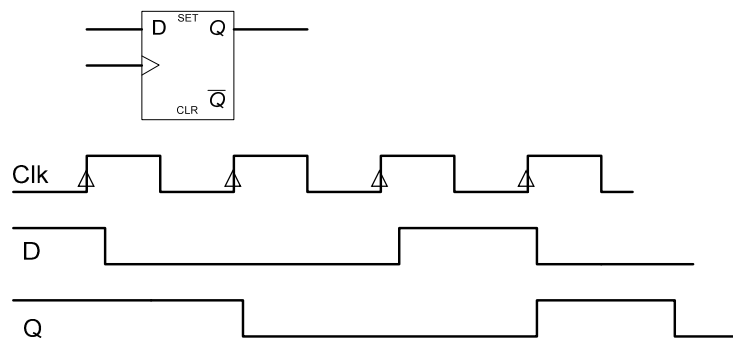
- Combinational logic
  - Output only depends on inputs
  - If you wait long enough you will get the right answer
- To build a processor, we also need to build sequential logic
  - Need to separate signals across clock cycles
  - They also provide temporary storage
  - This is usually done with flip-flops (or latches)
    - We will talk about flop-based design in this class

## Combinational Elements



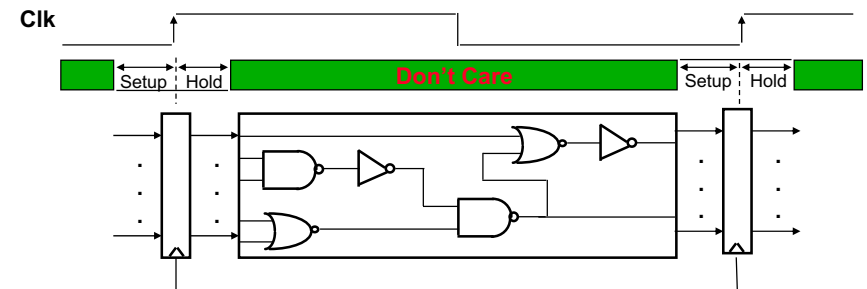
## D Flip Flops

- Samples its input on rising edge of clock
  - Holds the value it samples until next rising edge:



## Critical Timing Issues

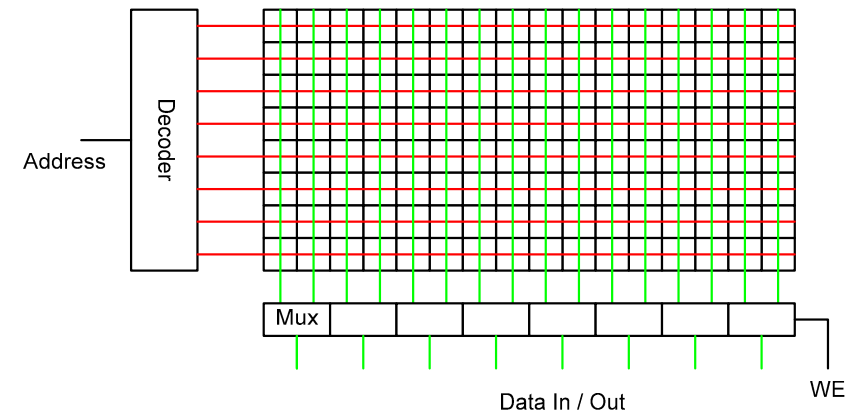
- Flops work great as long as input is stable when clock rises
  - Called setup and hold windows
  - Clock skew can cause some nasty problems
    - Hold time violations (we won't worry about this in this class)
- Cycle Time = Longest Prop Delay + Setup + Clock Skew



## Memory Structure

- Memory structures are generally specially designed
  - Could build them from flops or latches
    - But they would be big, slow, and power hungry
  - So circuit designers create the basic design
    - Create a module generator for logic designers to use

## Memory Diagram

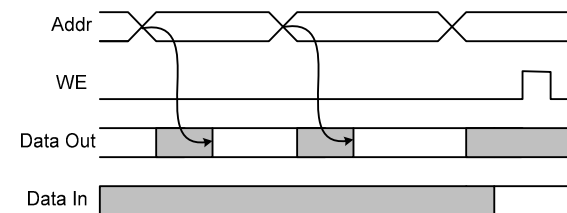


## Read from/Write to Memory

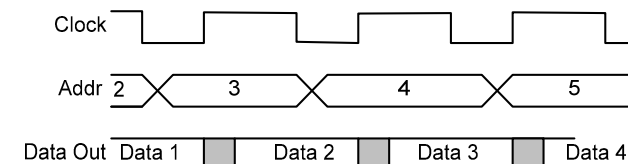
- Interface to Memory can be:
  - Combinational (asynchronous)
  - Clocked (synchronous)
- Combinational memory:
  - Read data is valid some delay after address lines settle
    - There is no clock.
  - Writes are tricky: must supply a write pulse in the middle of your address and data valid times
- Clocked memory (most common):
  - Memory looks like a standard synchronous device.
  - Address and control signals are sampled on rising edge of clock, and data is valid some number of cycles later

## Memory Timing

- Combinational/Asynchronous:

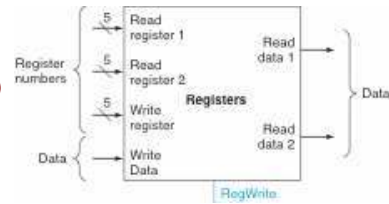


- Synchronous



## Memories In This Design

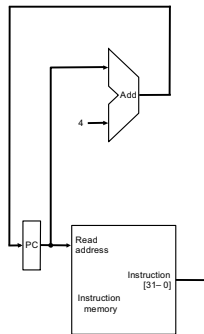
- They will be combinational
  - Otherwise we can't complete an instruction in one cycle!
- Interface is simple:
  - Inputs:
    - Address
    - DataIn
    - WriteEn (WriteEn must be a pulse)
  - Outputs:
    - DataOut
- Register file:
  - It has three address, two for reads, and one for write
  - It is called a 3-port, since it can perform 3 accesses per cycle



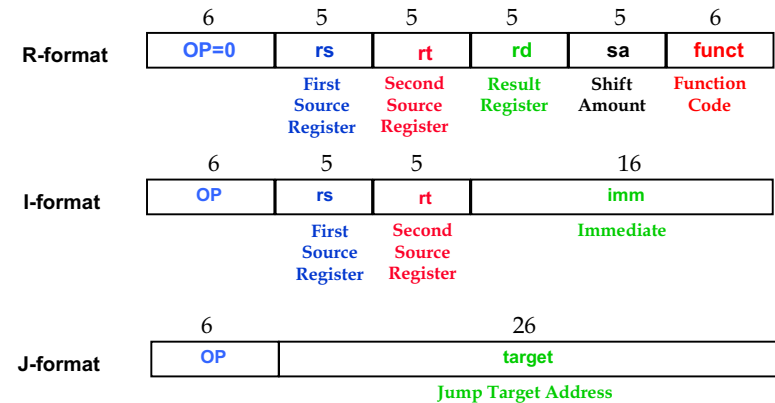
## The First Task: Fetching The Instruction (IF)

- Not that complex
- Instr = Mem[PC]
  - Fetch the instruction from memory
- Update program counter for next cycle
  - What is the address of the next instruction?

## Datapath: IF Unit



## What Did We Fetch?



## Nice Characteristics of MIPS Machine Code

- Instructions are fixed length
  - Don't need to decode first instruction to find next one
  - Always add 4 bytes to instruction pointer
- Register specifiers are always in the same place
  - Destination moves around some, but
  - Source registers are always in the same place
    - Or you don't need that register
  - Can fetch the registers BEFORE you decode instruction
    - Feed bits directly from the instruction memory

## Register to Register Operations

- In our subset this is only addu and subu
  - I did not want to worry about overflow

```
addu rd, rs, rt
subu rd, rs, rt
```

- Operation

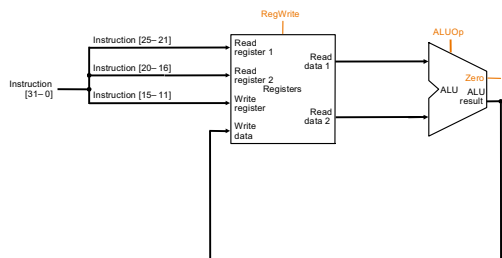
$R[rd] \leftarrow R[rs] + R[rt];$  Add operation

$R[rd] \leftarrow R[rs] - R[rt];$  Sub operation



## Datapath: Reg/Reg Ops

- $R[rd] \leftarrow R[rs] \text{ op } R[rt];$ 
  - ALU operation and RegWrite based on decoded instruction
  - Read Register 1, Read Register 2, and Write Register from rs, rt, and rd fields of instruction

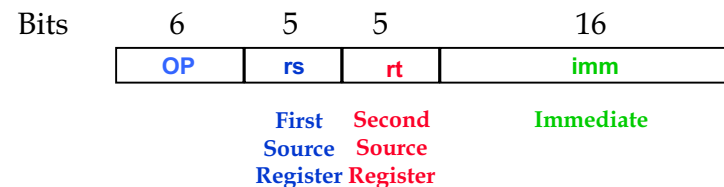


## OR Immediate RTL

- OR Immediate instruction
  - `ori rt, rs, imm`

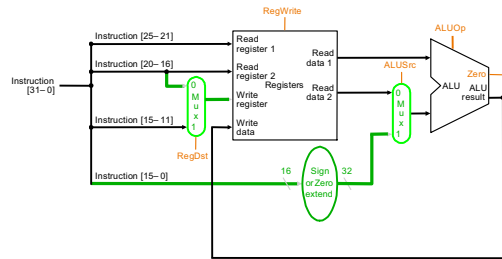
$R[rt] \leftarrow R[rs] \text{ OR } \text{ZeroExt}(imm);$

- Means I need to get instr[15:0] into the datapath, on RT path



## Datapath: Immediate Ops

- Extend datapath to support immediate operations
- Write register is rt or rd based on instruction
- Read data 2 is ignored for immediates
- Immediates can be sign or zero extended
- ALUsrc and ALU operation set based on instruction



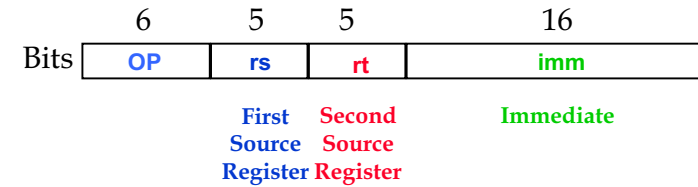
## Load

- Load instruction  
`lw rt, rs, imm`

$Addr \leftarrow R[rs] + \text{SignExt}(imm);$   
 $R[rt] \leftarrow \text{Mem}[Addr];$

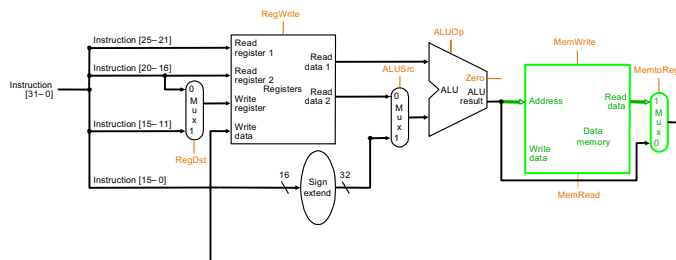
Compute memory address  
 Load data into register

- Notice this will use the immediate path as well



## Datapath: Load

- Extend datapath to support other immediate operations
- Extender handles either sign or zero extension
- MUX selects between ALU result and Memory output

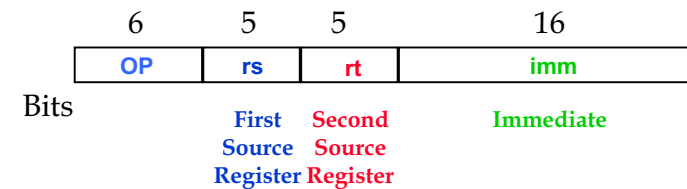


## Store

- Store instruction  
`sw rt, rs, imm`

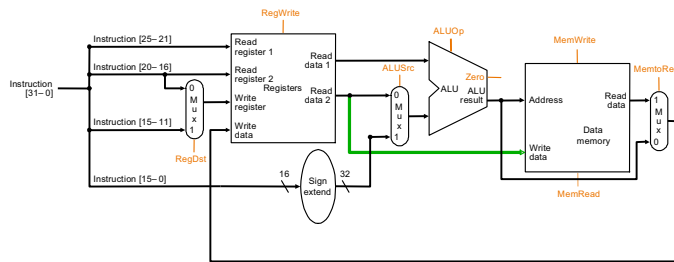
$Addr \leftarrow R[rs] + \text{SignExt}(imm);$   
 $\text{Mem}[Addr] \leftarrow R[rt];$

Compute memory addr  
 Load data into register



## Datapath: Store

- Read Register 2 is passed on to Memory
- Memory address calculated just as in lw case



## Branch

- Branch instruction
  - beq rs, rt, imm

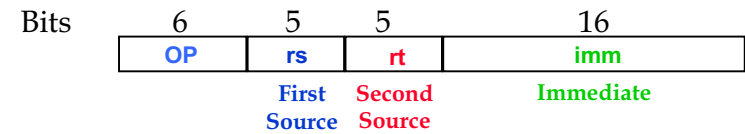
Cond <- R[rs] - R[rt];      Calculate branch condition  
if (cond eq 0)              Test if equal

PC <- PC + 4 + SignExt(imm)\*4

else

PC <- PC + 4;

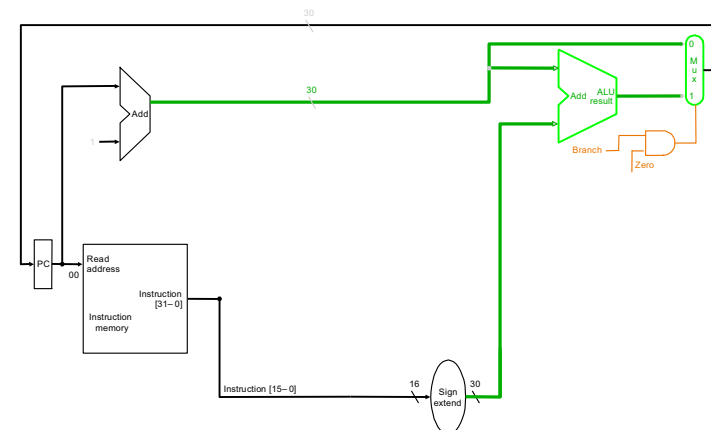
Calculate next address



## The Next Address

- PC is byte-addressed into instruction memory
  - Sequential
 
$$PC[31:0] = PC[31:0] + 4$$
  - Branch operation
 
$$PC[31:0] = PC[31:0] + 4 + \text{SignExt}(imm) \times 4$$
- Instruction Addresses
  - PC is byte addressed, but instructions are 4 bytes long
 
$$\Rightarrow \text{Simplify hardware by using 30 bit PC}$$
  - Sequential
 
$$PC[31:2] = PC[31:2] + 1$$
  - Branch operation
 
$$PC[31:2] = PC[31:2] + 1 + \text{SignExt}(imm)$$

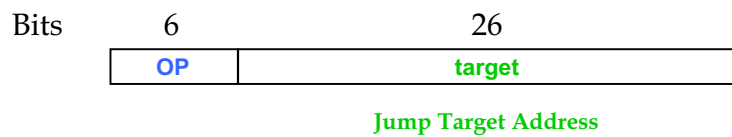
## Datapath: IF



## Jump RTL

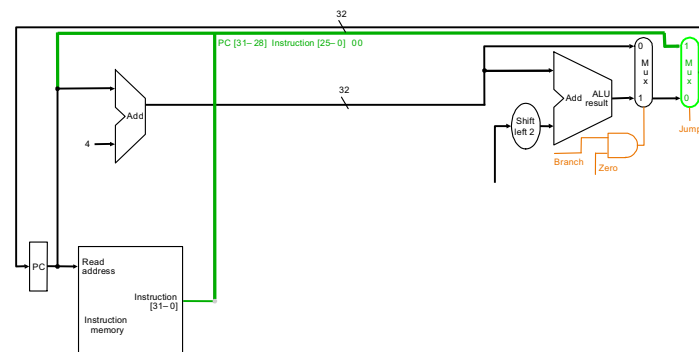
- Jump instruction  
j target

$PC[31:2] \leftarrow PC[31:29] \parallel target[25:0];$

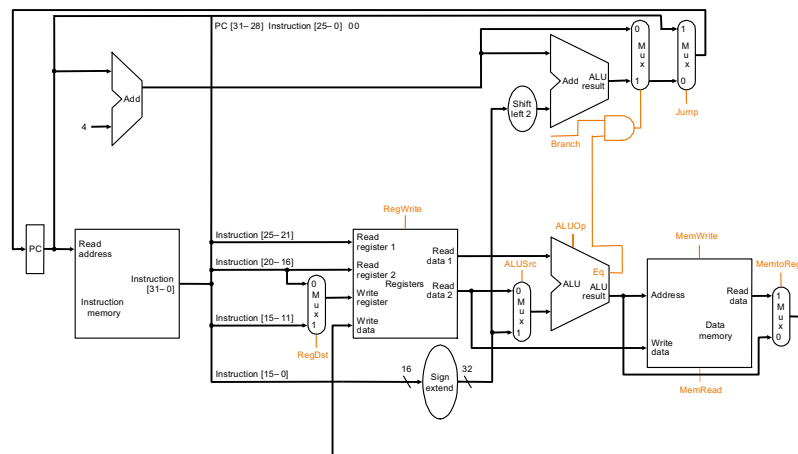


## Datapath: IFU with Jump

- MUX selects pseudodirect jump target



## Putting it All Together

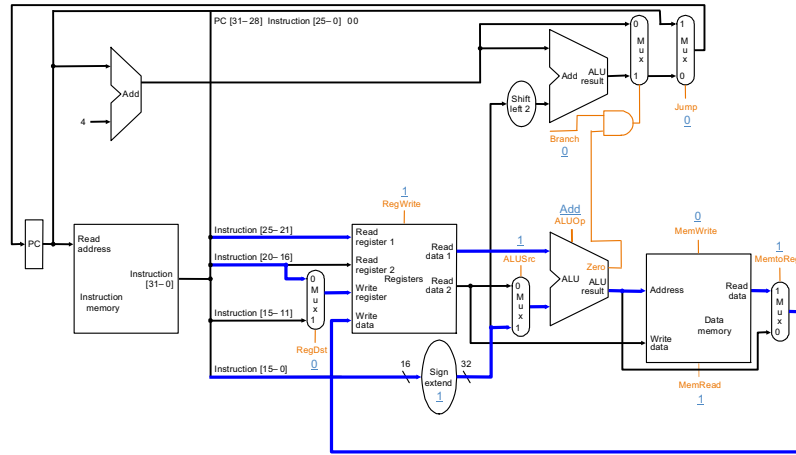


## Control

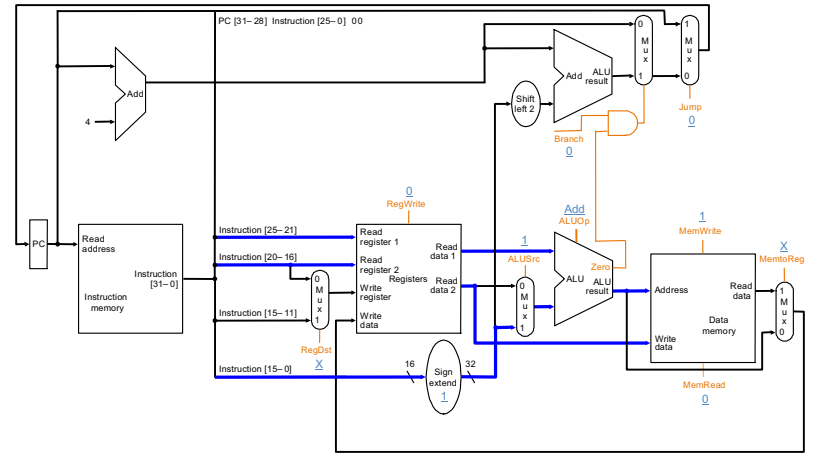
- Since every instruction takes one cycle, control is state free!
  - It is just decoded instruction bits
- There are also few control points
  - Control on the multiplexers
  - Operation type for the ALU
  - Write control on the Instruction & Data memories
- First part of cycle does not have any control
  - Which is good, since we don't have instruction yet
- Look at setting of the control points for different instructions



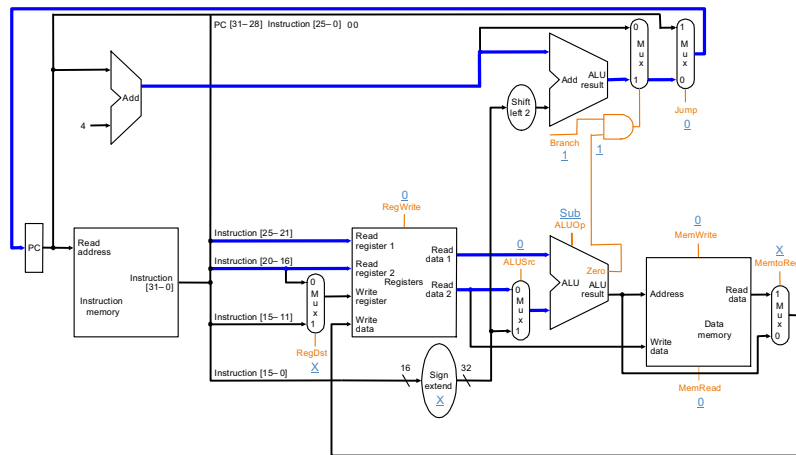
## Control for Load



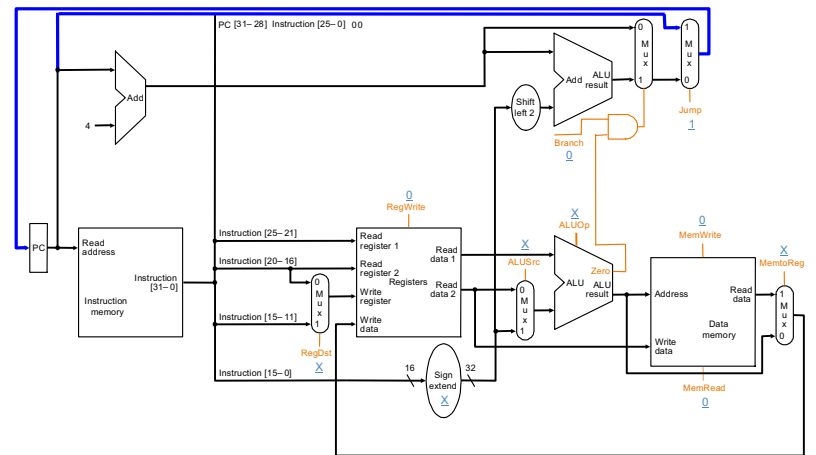
## Control for Store



## Control for Branch (beq)



## Control for Jump (j)

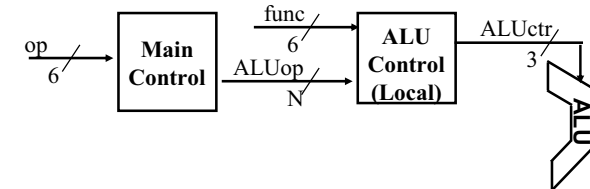


## Summary of Control Signals

func op	10 0000	10 0010	Not Important				
	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	add	sub	ori	lw	sw	beq	jump
RegDst	1	1	0	0	x	x	x
ALUSrc	0	0	1	1	1	0	x
MemtoReg	0	0	0	1	x	x	x
RegWrite	1	1	1	1	0	0	0
MemWrite	0	0	0	0	1	0	0
Branch	0	0	0	0	0	1	0
Jump	0	0	0	0	0	0	1
ExtOp	x	x	0	1	1	x	x
ALUctr<2:0>	Add	Sub	Or	Add	Add	Sub	xxx

## Multilevel Decoding

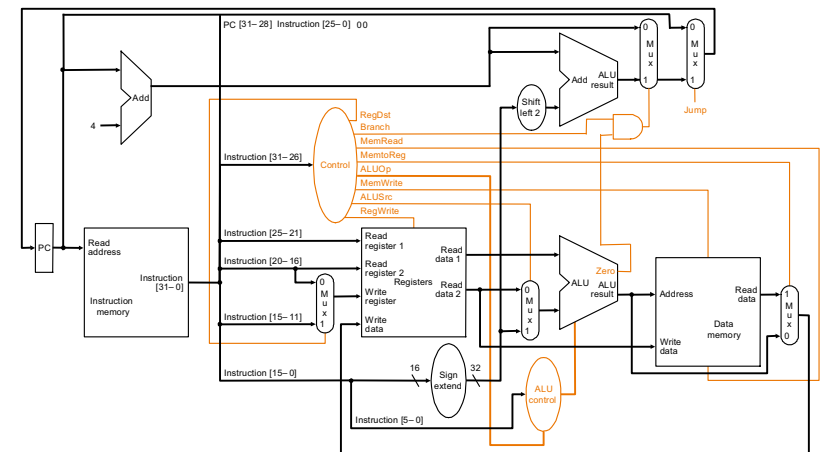
- Since only the ALU needs the func field
  - Pass it to the ALU unit, and have a local decoder there



## Multilevel Decoding (cont)

op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop<N:0>	"R-type"	Or	Add	Add	Subtract	xxx

## Putting It All Together



## Single Cycle Processor

---

- Advantages
  - Single cycle per instruction makes logic and clock simple
- Disadvantages
  - Inefficient utilization of memory and functional units since different instructions take different lengths of time
    - ALU only computes values a small amount of the time
  - Cycle time is the worst case path → long cycle times
    - Load instruction
  - All machines would have a CPI of 1

## Increasing Parallelism

---

- Problem:
  - Each functional unit used once per cycle
  - Most of the time it is sitting waiting for its turn
    - Well it is calculating all the time, but it is waiting for valid data
  - There is no parallelism in this arrangement
- Making instructions take more cycles makes machine faster!
  - Increases the parallelism going on in the machine
  - We will look at a 5 stage pipeline
    - Modern machines (Pentium 4) have order 20 cycles/instruction