Computer Organization and Design

CS251

Fall 2016
Rosina Kharal/Stephen Mann

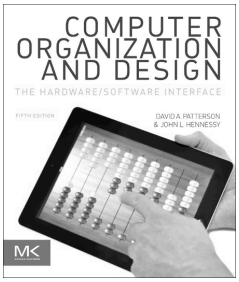
Course Texts/Course Notes

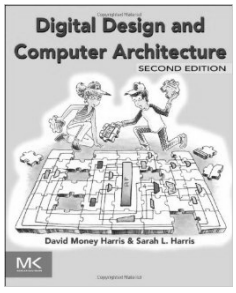
1. Patterson and Hennessy: **"Computer Organization and Design". 5th Edition**, Morgan Kaufmann, 2013, ISBN 0124077269.
4th Edition of the text also useful.

2.**(Optional)** Harris and Harris: "Digital Design and Computer Architecture". 2nd Edition, Morgan Kaufmann, 2012, ISBN 0123944244.

3.Course notes: Media.doc
First Half available

4.Texts also on reserve on DC Library for 3 hour loan





Grading Scheme: Course Webpage

- Assignments
  - 6 assignments each worth about 2.5% + bonus 1% for A0
- Clicker Participation (clicker questions)
- Midterm (October 25th 4:30)
- Final Exam (to be scheduled by the Registrar)

How to be Successful and Happy in this Course ☺

- Lectures are Essential: Try to Attend and be Awake
- Take Notes: In class discussion and examples and noting the correct answer to the clicker questions
- Do Assignments Yourself and as best as you can.
- Submit what you have.
- Midterm is usually well done. Final exam is challenging.
- Assignments 04, and 05 (now also 06) are critical in how well students perform overall.
- Do these assignments! Understand the solutions.
- There is a lot of material packed into the last month.
- Course Notes: General Overview
  - Not comprehensive and not a substitute for lectures

Office Hours

- Posted On Course Website
- Rosina begin with: **Thursday 12:30-1:30,** plus on the day assignments are due.

  *extra office hours during assignment weeks

  Stephen Mann : office hours will be posted.
- 2 Instructional Apprentices: Office hours will be posted
  - Wilson Hsu and Ahmed El-Roby
- Piazza: Posts answered usually in 24 hours **not 24 minutes ;)**

Assignments

- Assignments : Submitted online using Crowdmark
- Excessive collaboration is not allowed

Solutions in display case outside MC 4065

- Book authors request that solutions not be online
- Make sure you look at solutions while they are posted
  - Taking image of solutions is allowed.
  - Even a submission 5 minutes after the deadline is considered late.

How it All fits together...

- **If we tried to build an operational computer system from scratch**

It would certainly overwhelm us

- **Layers(operating system, microprocessor, memory, networking, wireless connectivity, input/output, etc.)**
- **Manage the layers of complexity: working one-by-one at individual layers of the Computer Architecture**
- **Understand each layer, and abstract away details of other layers that are not important to current layer**

Assembly Language: ***Defn from Wikipedia***

- "An assembly (or assembler) language,[1] often abbreviated ASM, is a low-level programming language for a computer, or other programmable device, in which there is a *very strong* (generally one-to-one) *correspondence between the language and the architecture's machine code instructions*. Each assembly language is specific to a particular computer architecture, in contrast to most high-level programming languages, which are generally portable across multiple architectures, but require interpreting or compiling. Assembly language may also be called symbolic machine code."

•**This Course:**

•Understanding of computer architecture, structure and evolution

•Computer architecture : instruction set architecture plus computer organization

• Instruction set architecture:

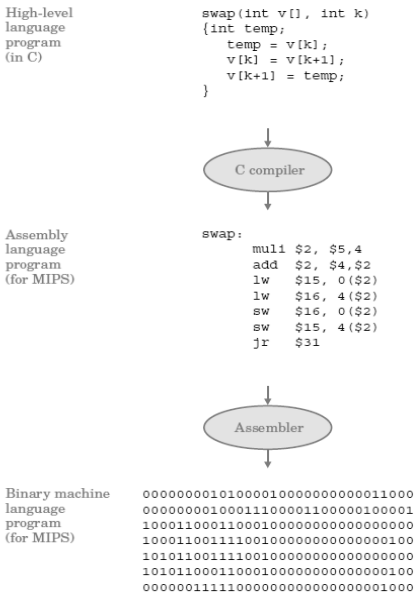
- Conceptual structure and functional behaviour of computing system as seen by programmer. How the processor works with native data types, low level instructions, registers, addressing modes, interrupt and exception handling, and input/output requests.

• Computer organization:

- Physical implementation, described in terms of functional units, their interconnection, how information flow among them is controlled

Course outline

*MIPS basic introduction (5 Instructions)
*Digital logic design
*Data representation and manipulation
*Designing a datapath
*Single-cycle datapath & Multicycle Datapath
*Pipelined datapath and pipelining hazards
*Memory Hierarchies (caches and virtual memory)
*Case Studies: VAX, SPARC, Pentium
*ARM : 32-bit



High Level Languages

Compiled or Interpreted
Eventually into Bytecode

These Instructions Execute in the data path hardware.
Instructions are interpreted piece by piece (bit by bit) ☺

About You 💻🕐

- A) I am CS major
- B) Math major
- C) Engineering
- D) Business or Science
- E) Other...

Computer Organization and Design

- **I want to take this course because:**
- A) I want to learn more about Hardware
- B) I want to learn about inner workings of computer systems
- C) I love CS
- D) I have to take this course- no choice
- E) I want to become a super awesome computer whiz- and save the world!

Other Courses I have This term

- **A) CS241 or CS246**
- **B) CS240 or CS245**
- **C) Stats**
- **D) Calculus or Algebra or C&O**
- **E) All of the above**

Guiding Principles : Computer Architecture Design

- **Use Abstraction to Simplify Design**
- Moores Law: Expect Rapid Change in Technology
  - IC resources doubles every 18-24 months
  - The number of transistors that can fit onto a circuit board will double. Design your architecture with this in mind...

Make the Common Case Fast

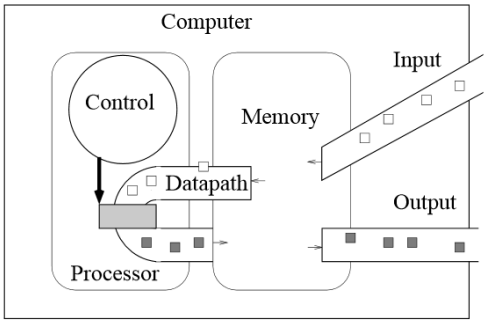
  - This will better enhance performance rather than optimizing the rare case. (common case is usually much simpler to optimize)

Guiding Principles : Computer Architecture Design

- **Improve Performance Via Parallelism**
  - Doing multiple tasks at once.
  - Divide and Conquer
- **Improve Performance Via Pipelining**
  - Prevalent in computer architecture design
- **Improve Performance Via Prediction**

  can be better to ask for forgiveness later than to ask for permission ☺

Big Picture



Instruction Set Architecture

- **To connect to the Hardware, you must speak its language**
  - Machine language
  - Send it machine level instructions to interpret and execute
- Different computers speak different dialects of the same language
  - MIPS: Instruction set from MIPS technologies 1980s
  - ARM : Similar to MIPS , iPad, iphones, A* series processors
  - Intel x86 Base architecture which extends the PC and much of Post-PC era

Similarities in all these Architectures

- **There must be many similarities in these Instruction sets**
  - Hardware underlying the systems are based on the same technological principles
  - There are common basic operations that all computers must be able to do very efficiently. Add, subtract, perform arithmetic operations. Working with floating point numbers.
  - Goals are similar: Maximize performance and minimize costs and energy. Try to get a completed instruction as fast as possible

Assembly Language such as MIPS:

- A**ssembly language** :**low-level**

programming language for a computer

- Programmers use to work much more at the Assembly level.
- Assembly language instructions translated into *Machine code* :
- Executed on the lower level machine : Processor
- Ordinary compiled code run 'directly' on the CPU , but programs do not run in vacuum. OS plays a critical role (system calls, access to I/O, DLL)

MIPS

- **What does MIPS stand for?**
- **Millions of Instructions per Second:**
- **Yes but that's not the MIPS we are interested in right now**

MIPS

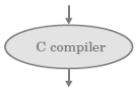
- **MIPS:**
- **Microprocessor without Interlocked Pipelined Stages….**
- **RISC architecture : Reduced Instruction Set**
  - **Very Basic operations**
  - **We look at Base MIPS 32 bit**

  More advanced versions of MIPS developed later

  ie) 64 bit etc.

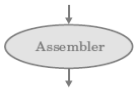
High-level language program (in C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```



Assembly language program (for MIPS)

```
swap:
      muli $2, $5,4
      add  $2, $4,$2
      lw   $15, 0($2)
      lw   $16, 4($2)
      sw   $16, 0($2)
      sw   $15, 4($2)
      jr   $31
```



Binary machine language program (for MIPS)

```
00000000101000010000000000011000
00000000100011100001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

High Level Languages

Compiled or Interpreted
Eventually into Bytecode

These Instructions Execute in the data path hardware.
Instructions are interpreted piece by piece (bit by bit) ☺

MIPS

Background Reading: 4th and 5th Edition 2.1-2.3
Computer Organization and Design

- **C code:**
  - **f = (g + h) – (i + j)**

Assume that variables f , g h and i and j are assigned to

Registers $s1….$s5 respectively

MIPS

- **C code:**
  - **f = (g + h) – (i + j)**

Assume that variables f , g h and i and j are assigned to

Registers $s1….$s5 respectively

Use two temp registers $s6 and $s7

- **$s1: f $s2: g $s3: h $s4: i $s5: j**

f= (g +h)- (i +j)

MIPS:

add $s6, $s2, $s3

add $s7, $s4, $s5

- **$s1: f $s2: g $s3: h $s4: i $s5: j**

f= (g +h)- (i +j)

MIPS:

add $s6, $s2, $s3

add $s7, $s4, $s5

sub $s1, $s6, $s7

MIPS: Overview

- Computers execute assembly instructions
- In binary on computer, but text form for people
- Only simple operations
- Addition, subtraction, goto, conditional goto
- Instructions operate on two types of data

– Registers—high speed access

– RAM—slow to access

- This course uses MIPS, which we review here
- Optional homework assignment on MIPS
- (1%: max 100% in this course ☺)

Registers:

- There are 32 registers
- Can use registers like a variable in a program, but via MIPS instruction
  - Each register has 32 bits, four bytes
  - $0, $1,...,$31
  - Sometimes: $s0,...,$s7, $t0,...,$t7
- Either okay, but don't mix $1 and $s1 in same program!
  - MIPS $0 always contains 0

Instruction Formats:

- Three general types of MIPS instructions
- Format refers to how many and what type of operands

R-Format: add $1,$2,$3

- Adds contents of $2 to contents of $3; store result in $1
- Often written as add *rd,rs,rt,* where *rd* is the destination

Register. Two source registers *rs* and *rt*.

I-Format: addi $1, $2, 100

- Adds immediate value 100 to contents of $2; store result in $1

J-Format: j 28

- Used for branching; discussed later

Memory:

- MIPS program can access 4 Giga-Bytes (4GB) of random
- access memory
- Memory accessed with number from 0 to 2^{32}-1
- Usually grouped in 4-byte blocks called words
- Most memory accesses are to addressed that are a multiple of 4
- Both MIPS program and data are stored in memory

Program in Memory:

- Each program instruction is one word in length
- Instruction address is multiple of four
- Often write a program as memory address followed by an instruction

```
Address:   Instruction
   100:    add $1,$2,$3
   104:    sub $1,$3,$5
   108:    addi $2,$12,16
```

- Often don't need address and use symbolic label of important instructions:

```
start: add $1,$2,$3
       sub $1,$3,$5
       addi $2,$12,16
```

Control Flow:

- In MIPS, no conditional statements like `if statement`
- In MIPS, no loop constructions such as `for`, `while`
- Control flow handled by *goto-like* commands

– jump (unconditional goto)
– beq (conditional goto)

- Special register, *program counter* (PC), stores address of executing instruction
- When non-goto instruction executed, PC incremented by 4
- This auto-increment advances the program to the next
- instruction

Jump:

jump: j

- `100: j 28`
- `104: add $1, $2, $3`
- `108: sub $1, $3, $5`
- `112: addi $2, $12, 16`
- When jump is executed, PC set to four times immediate argument

Conditional branch

- Example:

```
beq $1,$2,100
```

  - Compare contents of $1 to contents of $2
    If equal, add 4 times constant (100) and add to PC
    (×4 because constant is a word/instruction offset)
  - PC will also have 4 added to it
    PC updated to: PC+(4x100) + 4
  - If registers not equal, the instruction following branch is executed
- Constant can be negative
- `bne` similar but branches if values in registers are not equal

Conditional branch example

```
100: add $1, $0, $0
104: addi $2, $0, 6
108: addi $1, $1, 5
112: addi $2, $2, -1
116: bne $2, $0, -3
120: add $4, $6, $8
```

- Assume PC starts with value 100
- Fifth instruction (116) is conditional branch
  If contents of $2 not equal to zero, we branch to...

Memory access

- 32 registers clearly not enough to store data of most programs
- Special MIPS instructions to access 4GB RAM
- All memory accesses handled by two I-Format instruction
- Load word: Reads word from memory, stores in register

```
100:    lw $1, 100($2)
```

  Read value stored at memory address 100+$2 (M[100+$2]) store result in register $1
- Store word: Takes value of register and write it to memory

```
100:    sw $1, 100($2)
```

  Write the value in register $1 to M[100+$2]
- This is word address, so 100+$2 must be a multiple of 4

Performance

- How can we compare different computer designs?
- Two important measures of performance:
  - Response time: time between start and completion of a task
  - Throughput: total amount of work done in a given time
- Improving response time usually improves throughput
- Analogy: grocery-store checkout
- Nearly all computers have a clock
- Useful concepts: clock ticks, cycles, clock rate

Performance

- Individual Computer
  - Response Time: Time between start and completion of a task
    - Also known as ***Execution Time***
- Datacenter Manager:
  - Several servers running inquiries submitted by many users
  - Throughput: Total amount of work done in a given time

Response Time/ Throughput

Would the following improvements improve the Response time, Throughput or both:

- Changing the processor to a newer and faster version

- Adding in processors to a system, thereby allowing multiple tasks to begin execution at the same time. (each processor is dedicated to an individual task)

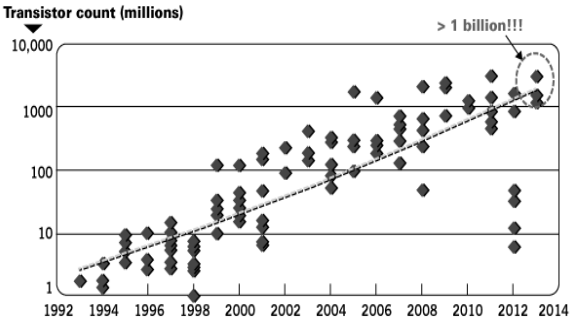
Response Time/ Throughput

Would the following improvements improve the Response time, Throughput or both:

- Changing the processor to a newer and faster version
- improves response time and increase in throughput:
- More tasks able to complete
- Adding in processors to a system, thereby allowing multiple tasks to begin execution at the same time. (each processor is dedicated to an individual task)
- Although execution time of individual tasks remains the same- response time likely improved do to multiple processors (multiple check-outs)
- Throughput increase: More tasks completed in a given amount of time

Uniprocessors to Multiprocessors

"Moore's Law" (Moore's Trend) describes the tendency for the number of transistors that can be place on an integrated circuit board to double approximately every two years.



Design Gap:
Are we able to maximize the use
Of these advents in technology

Leveling off:
Have we now reached or come close to reaching a Plateau.

Power Consumption : Improvements are not
At the same rate

Articles to Read:

http://www.maltiel-consulting.com/ISSCC-2013-High-Performance-Digital-Trends.html
http://www.extremetech.com/computing/190946-stop-obsessing-over-transistor-counts-theyre-a-terrible-way-of-comparing-chips

Example: Registers

```
#include<stdio.h>
#define NR 10000
#define NC 10000

int a[NR][NC];

void main() {
   int i,j;
   for (i=0;i<NR;i++){
      for (j=0;j<NC;j++){
       ;
      }}}
```

- register int i, j: 0.044 sec
- int i,j: 0.27 sec

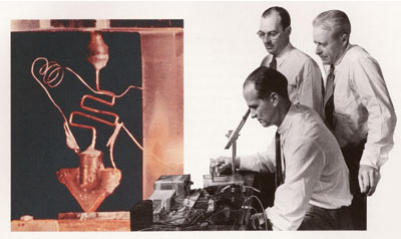
(approx 6 times slower!)

Basics of Digital Logic Design

Transistors

Binary digits

- Basic unit of information in digital communication: 0 or 1
- Represented internally as a high or low voltage
- Transistors : An on/off switch controlling the flow of electricity
- **CS251: Introduce basic MOS transistors in order to understand how basic gates can be implemented via transistors.**



- **The first transistor was invented in 1947 by William Shockley John Bardeen and Walter Brattain at Bell Laboratories.**
- **Most important electronics event of the 20th century: leading to IC (integrated circuit) and microprocessor technology serving as the basis of electronics**

Image source: http://www.cedmagic.com/history/transistor-1947.html

Background on Transistors

- First Early Computer: Charles Babbage and Early form of Computers
  - Used Relays or Vacuum Tubes to represent states of matter
- Transistors : Cheap, Reliable and Small
  - Voltage applied to a control terminal (On or Off is generated)
- 1959: Robert Noyce: (Cofounded Fairchild Semiconductor and Intel) patented a method of interconnecting many transistors on a single chip.
  - At that time transistors cost was ~$10 ☺

Background on Transistors

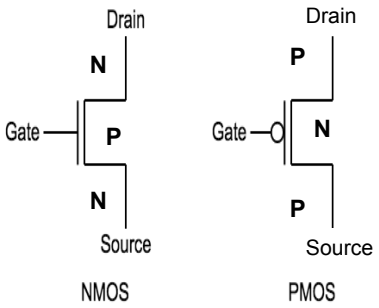
- First Early Computer: Charles Babbage and Early form of Computers
  - Used Relays or Vacuum Tubes to represent states of matter
- Transistors : Cheap, Reliable and Small
  - Voltage applied to a control terminal (On or Off is generated)

- 1959: Robert Noyce: (Cofounded Fairchild Semiconductor and Intel) patented a method of interconnecting many transistors on a single chip.
  - At that time transistors cost was ~$10 ☺
  - Today we pack ~1 billion transistors onto a chip
  - Lucky the cost went down… ~10 *micro-cents* a transistor

MOSFETS : Metal-Oxide-semiconductor field effect transistors
Also Known As : MOS transistors

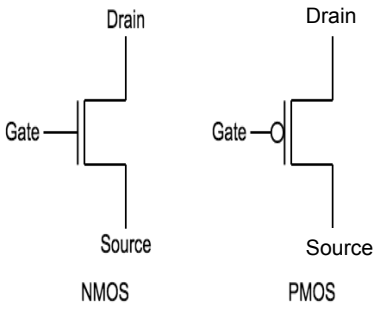
- Built from silicon (Si) : 4 valence electrons in outer shell
- Silicon forms strong crystal lattice:
- All electrons tied up in adjacent bonds →Therefore poor conductor
- However, (Si) becomes a better conductor when small amounts of impurities (called dopants) are added to this lattice structure
- **N-Type**: Small amounts of Arsenic(5 valence electrons) atoms added – leaving extra floating electrons that are not involved in bonding. *Negatively charge particles*
- **P-Type**: Small amounts of Boron(3 valence electrons) atoms added – leaving positively charged ions: *Positively charged particles*
- Silicon becomes a Semiconductor with the addition of dopants

MOS Transistors: NMOS and PMOS



- MOS transistors :sandwich of layers of conducting and insulating materials
- nMOS (npn) and pMOS (pnp)

MOS Transistors: NMOS and PMOS



- MOS transistors :sandwich of layers of conducting and insulating materials
- nMOS (npn) and pMOS (pnp)
- Labeled: **Gate-Source-Drain**
- **Two types of Transistors have opposite behaviours**
- **That compliment each other: CMOS uses both nMOS and pMOS**

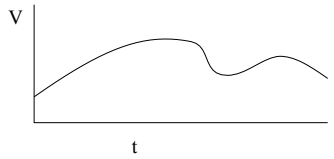
- Further Reading
- **http://hyperphysics.phy-astr.gsu.edu/hbase/solids/dope.html#c4**
- http://www.cs.mun.ca/~paul/transistors/node1.html

A Brief Look at Electricity

- Computers work with current/voltage

$V = IR$

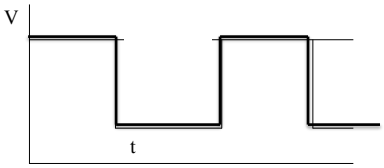
- These quantities are continuous Plot: voltage vs time



- They can be manipulated, but accuracy is difficult

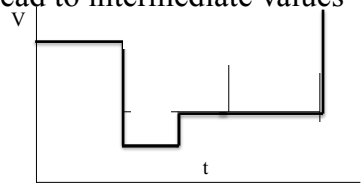
Digitizing

- Discrete Signal



Direct Current: Current flows in one direction through a circuit

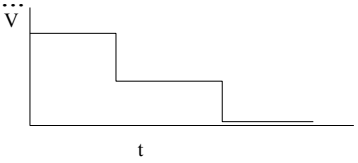
- Signal is either high (1) or low (0)
- Transformation could lead to intermediate values



but these can be "designed out"

Why Binary?

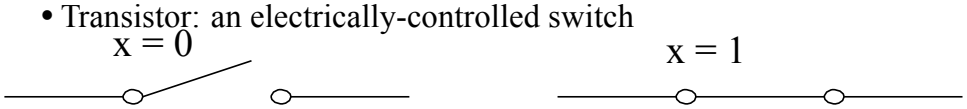
- Could have more levels...



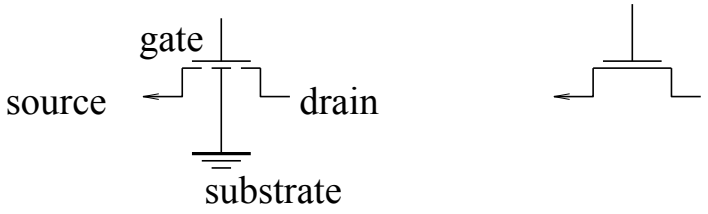
- High, Medium, Low (2,1,0 – ternary)
- Two levels are simpler and just as expressive
- Nearly all computers today use binary
- Nearly all computers have similar underlying structure

Implementing Gates Using Transistors

- Transistor: an electrically-controlled switch

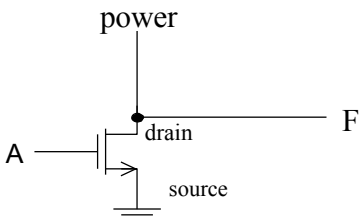


- An NMOS transistor ("n-transistor") and its symbol



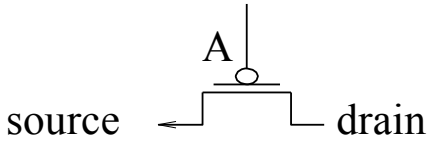
- This behaves like the switch above
- We have a 1 or 0 coming out of our switch

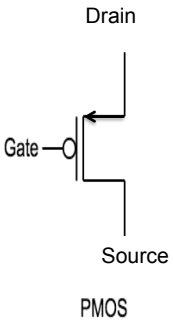
An NMOS NOT



- Source Is connected to ground (0 volts)
- If $A = 1$, then low resistance between drain and source ($F = 0$)
- Problem: Surge of current to the ground
- If $A = 0$, then very high resistance between drain and source ($F = 1$)
- Problem: transmits strong 0 but weak 1

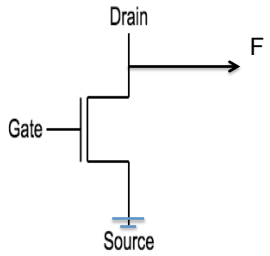
A PMOS transistor

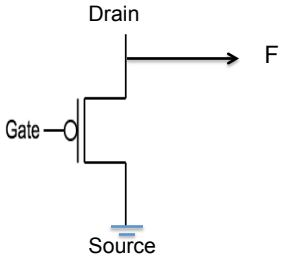




- Opposite behaviour to NMOS:
  - If $A = 1$, high resistance between drain and source
  - If $A = 0$, low resistance between drain and source
  - Problem: Transmits strong 1 but weak 0
- Denote inversion with "bubble"

Simple Transistor Analysis





NMOS

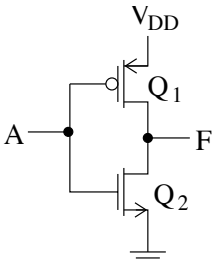
Input	$A = 0$	$A = 1$
Resistance	High	Low

PMOS

Input	$A = 0$	$A = 1$
Resistance	Low	High

- To analyze CMOS circuit:
  - Make table with inputs, transistors, and output(s)
  - For each row of table (setting of inputs), check whether transistor resistance is High,Low
  - For each row of table, check if output has clean path to power (1) ground (0)

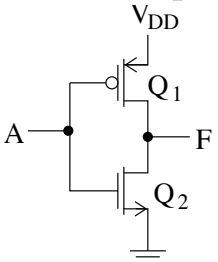
Complementary MOS Transistor (CMOS)



PMOS and NMOS together form CMOS: Compliment MOS

WHY:
We want both 0, 1 signals to be strong readings
Use properties of NMOS and CMOS together

Complementary MOS Transistor (CMOS)



PMOS and NMOS together form CMOS:

Input signal A goes into the Gate at both Transistors

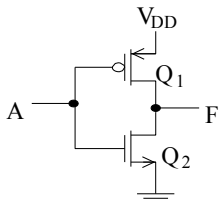
*Q_1, Q_2 determine connection to ground
*Connected to V_{DD} Power

A	Q_1	Q_2	F
0	Low	High	1
1	High	Low	0

➔That will determine F

CMOS

- CMOS circuits use both n-transistors and p-transistors
- Will build circuits with "clean" paths to exactly one of power and ground.
- Q_1 Q_2 are the inner resistances of each transistor

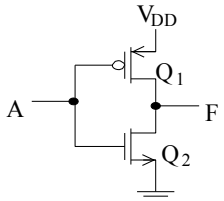


A	Q_1	Q_2	F
0	Low	High	1
1	High	Low	0

- No bad flow of current from power to ground
- No weak transmissions

CMOS

- CMOS circuits use both n-transistors and p-transistors
- Will build circuits with "clean" paths to exactly one of power and ground.
- Q_1 Q_2 are the inner resistances of each transistor

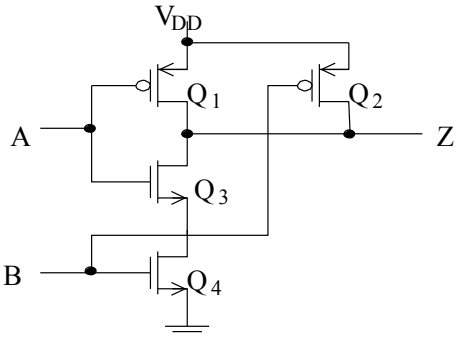


CMOS NOT:
The Output is the Inverse of the input A.

A	Q_1	Q_2	F
0	Low	High	1
1	High	Low	0

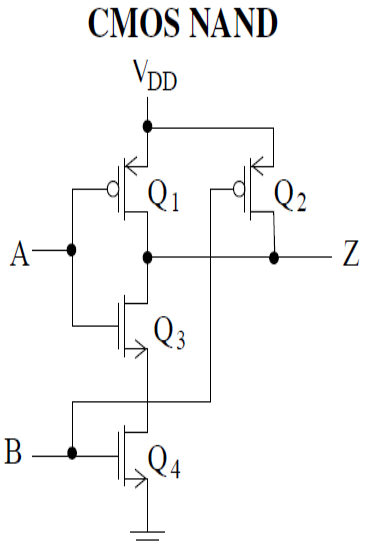
- No bad flow of current from power to ground
- No weak transmissions

CMOS NAND



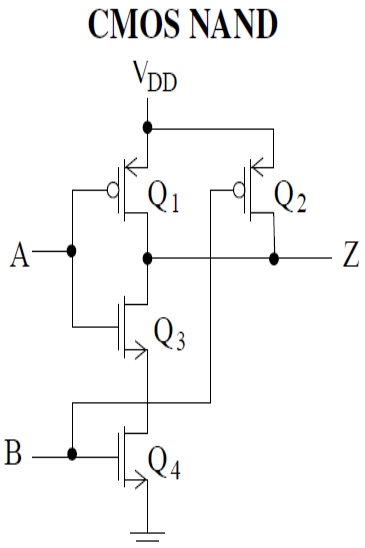
A	B	Q_1	Q_2	Q_3	Q_4	Z
0	0	Low	Low	High	High	1
0	1					
1	0					
1	1					

2-input NAND to a 3-input NAND



How do we scale this to add more Input sources?

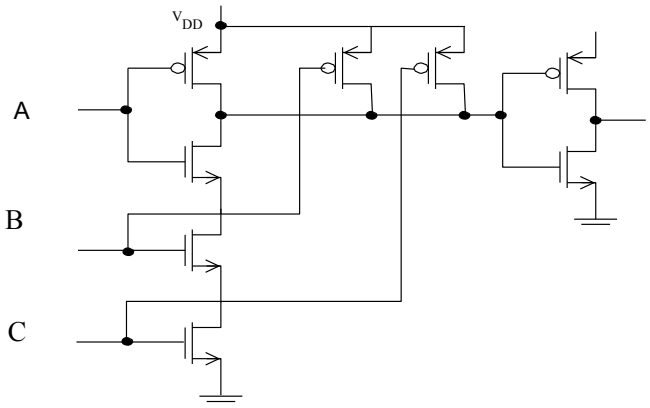
2-input NAND to a 2-input ***AND Gate***



How can we change this to an AND Gate?

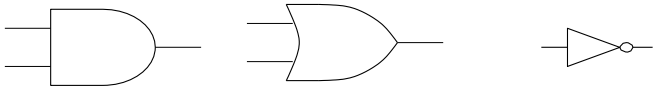
CMOS AND and OR

- To get AND and OR, add inverter at end
- Example: 3 Input AND



- Thus, NAND is preferred to AND in actual circuits

Transistors Implement The Basic Building Blocks of Circuits



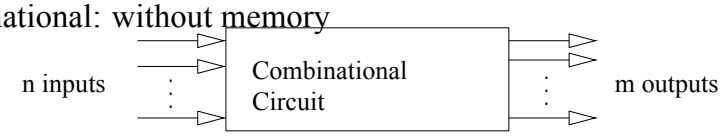
- AND, OR, NOT Gates
- NOT is often drawn as "bubble" on input or output
- AND, OR can be generalized to multiple inputs (useful)
- We can design using AND, OR, NOT, and optimize hardware afterwards
- In practice, at the transistor level we work with NAND or NOR gates

Basic Digital Logic Design

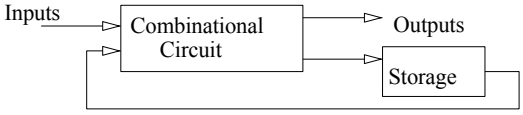
Truth Tables and Digital Building Blocks

Logic Blocks

- Readings: Appendix B, sections B.1–B-3, B.7-10. (C in other versions)
- Combinational: without memory



- Sequential: with memory



- Inputs and outputs are 1/0 (High/low voltage, true/false)

Specifying input/output behaviour

- Truth table: specifies outputs for each possible input combination

X	Y	Z	F	G
0	0	0	0	1
0	0	1	1	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	0

- Complete description, but big and hard to understand

Compact alternative: Boolean algebra

- Variables (usually *A, B, C* or *X, Y, Z*) have values 0 or 1
- OR (+) operator has result 1 iff either operand has value 1
- AND (⋅) operator has result 1 iff both operands have value 1
  $A \cdot B$ often written AB
- NOT (¬) operator has result 1 iff operand has value 0
  $\neg A$ usually written $\overline{A}$

OR		
A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

AND		
A	B	AB
0	0	0
0	1	0
1	0	0
1	1	1

NOT	
A	¬A
0	1
1	0

- For truth table on previous slide, clearly $G = \overline{XYZ}$
- $F = \overline{X}\,\overline{Y}Z + X\overline{Y}\,Z + XY\overline{Z} + XYZ$ (not obvious)

Truth Table to Formula Using Minimal Terms

A	B	C	F	$\bar{A}\bar{B}C$	$A\bar{B}C$	ABC	$\bar{A}\bar{B}C+A\bar{B}C+ABC$
0	0	0	0	0	0	0	0
0	0	1	1	1	0	0	1
0	1	0	0	0	0	0	0
0	1	1	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	0	1	0	1
1	1	0	0	0	0	0	0
1	1	1	1	0	0	1	1

Two-Level Representations

- Any Boolean function can be represented as a sum of products (OR of ANDs) of literals
- Each term in sum corresponds to a single line in truth table with value 1
- This can be simplified by hand or by machine
- Product of sums representation may also be useful

Don't Cares in Truth Tables

- Represented as X instead of 0 or 1
- When used in output, indicates that we don't care what output is for that input
- When used in input, indicates outputs are valid for all inputs created by replacing X by 0 or 1 (useful in compressing truth tables)
- Example:

A	B	C	F
0	0	X	0
0	1	X	1
1	X	X	X

Compressed Truth Tables and Non-Minimal Terms

A	B	C	F	$\bar{A}\bar{B}C$	AC	$\bar{A}\bar{B}C+AC$
0	0	0	0	0	0	0
0	0	1	1	1	0	1
0	1	X	0	0	0	0
1	X	0	0	0	0	0
1	X	1	1	0	1	1

Using Overlapping Non-Minimal Terms

A	B	C	F	AB	AC	$AB+AC$
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	0
1	0	0	0	0	0	0
1	0	1	1	0	1	1
1	1	0	1	1	0	1
1	1	1	1	1	1	1

Laws of Boolean Algebra

Rule	Dual Rule	
$\overline{\overline{X}} = X$		
$X+0=X$	$X \cdot 1 = X$	(identity)
$X+1=1$	$X \cdot 0 = 0$	(zero/one)
$X+X=X$	$XX=X$	(absorption)
$X+\overline{X}=1$	$X\overline{X}=0$	(inverse)
$X+Y=Y+X$	$XY=YX$	(commutative)
$X+(Y+Z)=(X+Y)+Z$	$X(YZ)=(XY)Z$	(associative)
$X(Y+Z)=XY+XZ$	$X+YZ=(X+Y)(X+Z)$	(distributive)
$\overline{X+Y}=\overline{X}\cdot\overline{Y}$	$\overline{XY}=\overline{X}+\overline{Y}$	(DeMorgan)

- DeMorgan's Law:

$$\overline{X+Y} = \overline{X} \cdot \overline{Y} \qquad \overline{XY} = \overline{X} + \overline{Y}$$

First we will look at Logic Gates: How DeMorgan's is implemented via Gates

- Distributive Law:
- X + YZ = (X + Y)(X + Z) why?
- =XX + XZ + XY + YZ
- Factor out X :
- =X (X +Z +Y) + YZ

If X is false

- Distributive :
- X + YZ = (X + Y)(X + Z) why?
- =XX + XZ + XY + YZ
- Factor out X :
- =X (X +Z +Y) + YZ

Just X being True is enough
To make the first part of the expression true.
Therefore bracketed term drops

- Distributive :
- X + YZ = (X + Y)(X + Z) why?
- =XX + XZ + XY + YZ
- Factor out X :
- =X+ YZ

Just X being True is enough
To make the first part of the expression true.
Therefore bracketed term drops

- Distributive :
- X + YZ = (X + Y)(X + Z) why?
- =XX + XZ + XY + YZ
- Factor out X :
- =X+ YZ

Just X being True is enough
To make the first part of the expression true.
Therefore bracketed term drops

Binary Numbers: 0010 ? Binary Number system: Review

Formula Simplification Using Laws

- We can use algebraic manipulation (based on laws) to simplify formulas
- An example using the previous truth table

$$\begin{aligned} F &= \overline{X}\,\overline{Y}\,Z + X\overline{Y}\,Z + XY\overline{Z} + XYZ \\ &= \overline{Y}\,Z(\overline{X} + X) + XY(\overline{Z} + Z) \\ &= \overline{Y}\,Z + XY \end{aligned}$$

X	Y	Z	F	G
0	0	0	0	1
0	0	1	1	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	0

- Now with F we have a boolean formula.
- Simplifying is tricky even for humans.
- Hard to automate
- Is simplest formula the best for implementation?

Formula Simplification Using Laws

- We can use algebraic manipulation (based on laws) to simplify formulas
- An example using the previous truth table

$$\begin{aligned} F &= \overline{X}\,\overline{Y}\,Z + X\overline{Y}\,Z + XY\overline{Z} + XYZ \\ &= \overline{Y}\,Z(\overline{X} + X) + XY(\overline{Z} + Z) \\ &= \overline{Y}\,Z + XY \end{aligned}$$

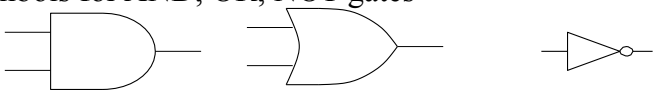
X	Y	Z	F	G
0	0	0	0	1
0	0	1	1	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	0

- The boolean expression at any level can be implemented with AND / OR and NOT gates

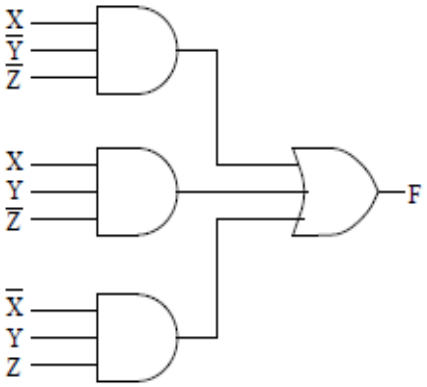
What was a formula for G?

Using Gates in Logic Design

- Here are symbols for AND, OR, NOT gates



- NOT often drawn as “bubble” on input or output
- AND, OR can be generalized to many inputs (useful)
- We can design using AND, OR, NOT, and optimize afterwards
- In practice, logic minimization software works with NAND or NOR gates, or at transistor level

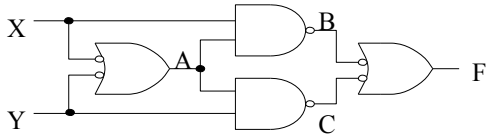


Good

$F = X\overline{Y}\overline{Z} + XY\overline{Z} + \overline{X}YZ$

Deriving Truth Table from a Circuit

- Label intermediate gate outputs
- Fill in truth table in appropriate order

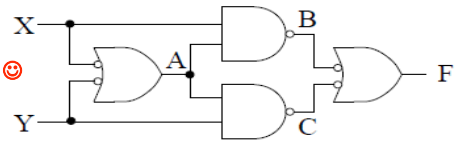


X	Y	A	B	C	F
0	0				
0	1				
1	0				
1	1				

Deriving Truth Table from Circuit

- Label intermediate gate outputs
- Fill in truth table in appropriate order

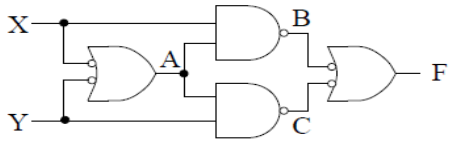
Inverted Inputs ☺



X	Y	A	B	C	F
0	0	1			
0	1	1			
1	0	1			
1	1	0			

Deriving Truth Table from Circuit

- Label intermediate gate outputs
- Fill in truth table in appropriate order

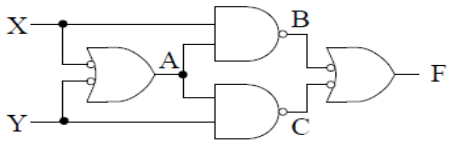


What Gate is this

X	Y	A	B	C	F
0	0	1	1	1	0
0	1	1	1	0	1
1	0	1	0	1	1
1	1	0	1	1	0

Deriving Truth Table from Circuit

- Label intermediate gate outputs
- Fill in truth table in appropriate order



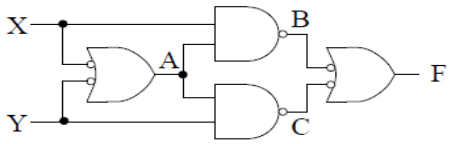
X	Y	A	B	C	F
0	0	1	1	1	0
0	1	1	1	0	1
1	0	1	0	1	1
1	1	0	1	1	0

What Gate is this

XOR : exclusively 1 input or the other Not Both

Deriving Truth Table from Circuit

- Label intermediate gate outputs
- Fill in truth table in appropriate order



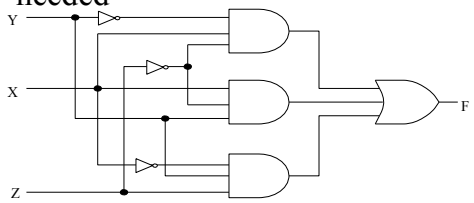
Note this is an alternative way of examining this circuit.
Both are correct.

This method you use
Intermediate gates
As AND gates
And then B,C values
Are double inverted
Which cancels out.

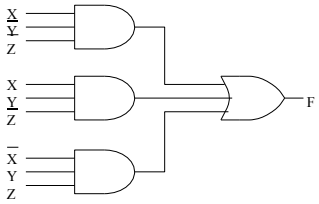
X	Y	A	B	C	F
0	0	1	0	0	0
0	1	1	0	1	1
1	0	1	1	0	1
1	1	0	0	0	0

Good Style in Circuit Drawing

- Assume all literals (variables and their negations) are available
- Rectilinear wires, dots when wires split
- Do not draw spaghetti wires for inputs; instead, write each literal as needed



Bad



Good

F = _____

Useful Components: Decoders

- n inputs, 2^n outputs (converts binary to "unary")
- Example: 3-to-8 (or 3-bit) decoder

			Outputs							
A_2	A_1	A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Useful Components: Decoders

- n inputs, 2^n outputs (converts binary to "unary")
- Example: 3-to-8 (or 3-bit) decoder

			Outputs							
A_2	A_1	A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Only **one** output asserted for each Input combination

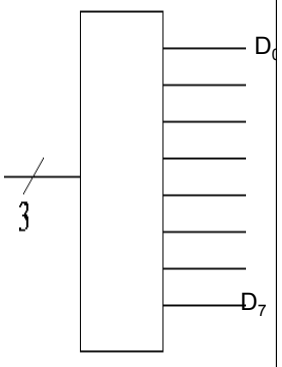
Decoder: Translates the n-bit input into a signal that corresponds to the binary value of the n-bit input
A useful component in building larger ICs

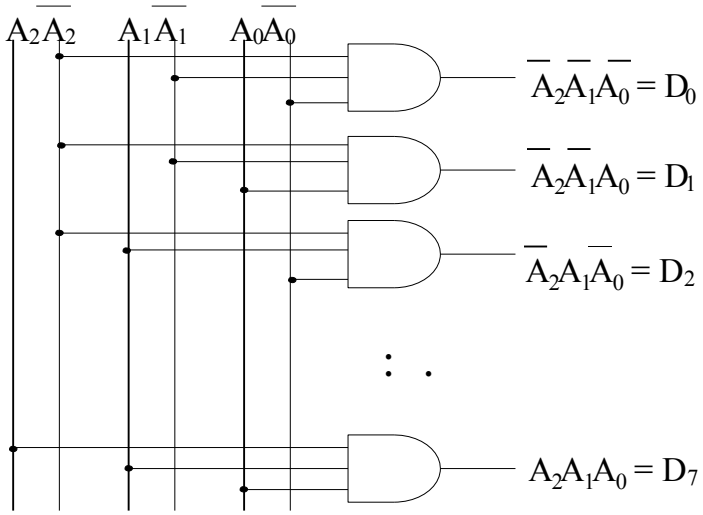
Useful Components: Decoders

- n inputs, 2^n outputs (converts binary to "unary")
- Example: 3-to-8 (or 3-bit) decoder

			Outputs							
A_2	A_1	A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

- Circuit has regular structure





All Outputs are expressed as minterms from the Truth Table.

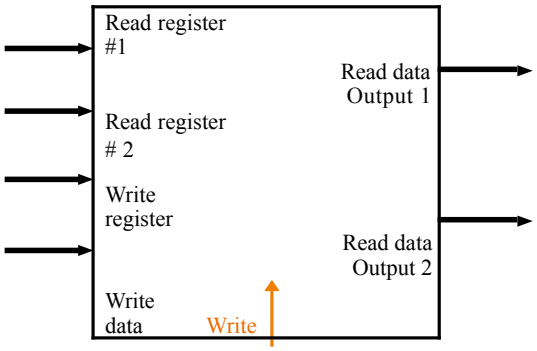
Easy to see how AND gates implement Each minterm.

No OR gate Needed ☺

3-to-8 (or 3-bit) Decoder

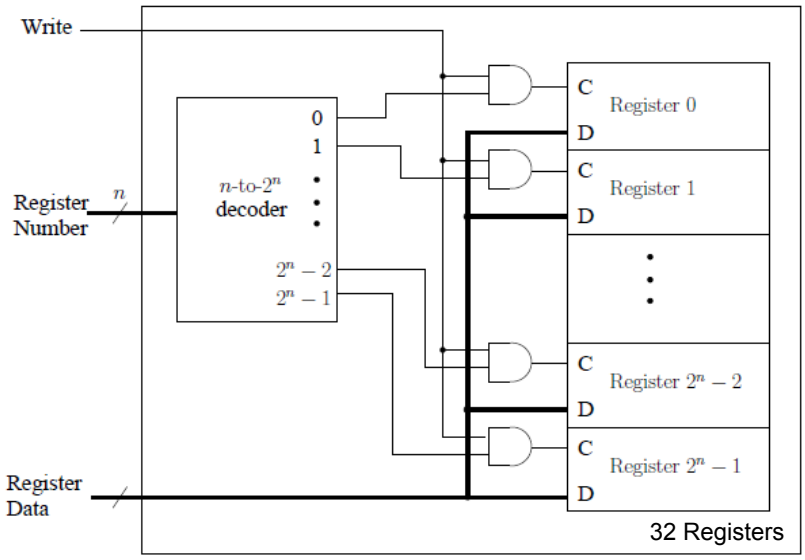
Registers and Register Files

- Register: an array of flip-flops (we will discuss soon)
- Register file: a way of organizing registers



Write Bit turned on for writing

Register File : Write Operation



Writing Data to a Register

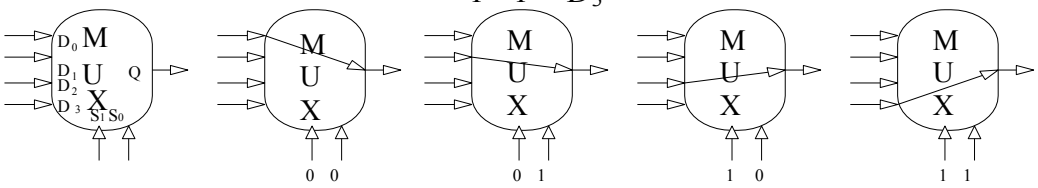
Using a Decoder

How does this work?

Multiplexors

- Inputs: 2^n lines ($D_0, \ldots, D_{2^n - 1}$)
  n select lines ($S_{n-1}, \ldots, S_0$)
- Output: The value of the D_S line
- Example: 4-1 Multiplexer

S_1	S_0	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3



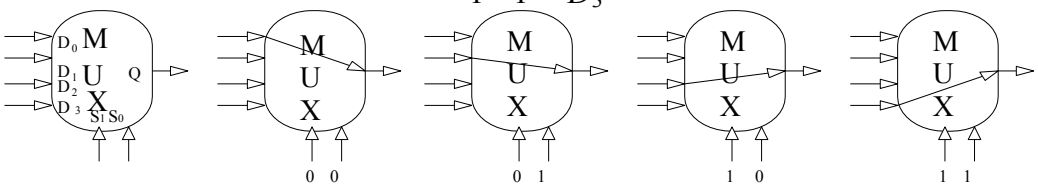
Multiplexors

- Also called a selector:
- The output is only **one** of the inputs
- Which input gets through

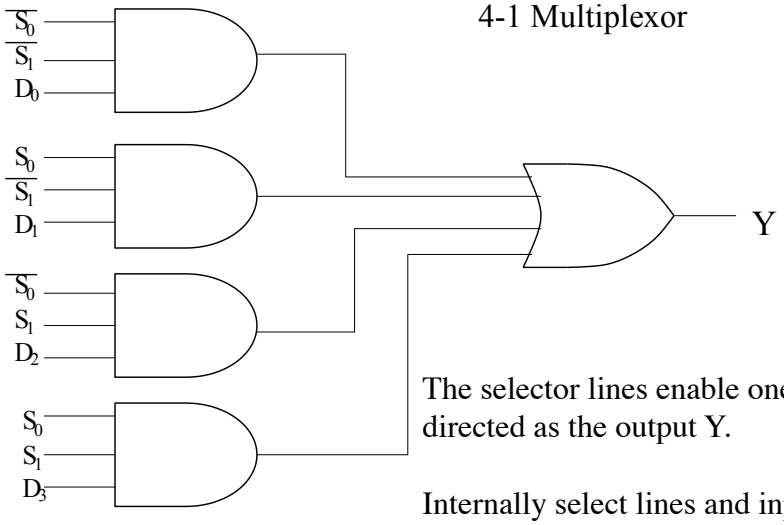
Is decided by select lines.

S_1	S_0	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

- This is 4x1, can have 2x1, 8x1 etc.



4-1 Multiplexor

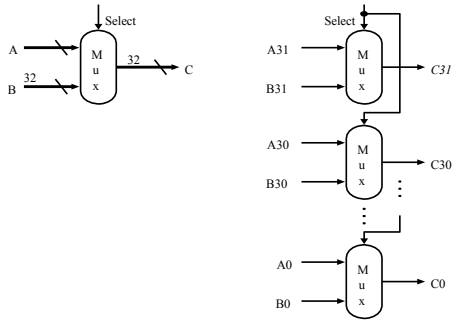


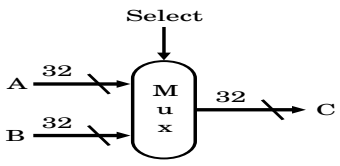
The selector lines enable one of D_0 to D_3 to be directed as the output Y.

Internally select lines and inputs are tied to AND gates

Arrays of Logic Elements

- "Slash" notation is used to indicate lines carrying multiple bits, and to imply parallel constructions



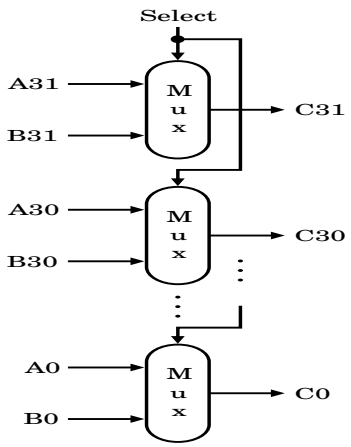


Slash indicates a 32 bit line Bus

Bus: Collection of data lines that are treated together as a single logical signal.

Multiplexor: Allows us to select one bus or data source over another

a. A 32-bit wide 2-to-1 multiplexor

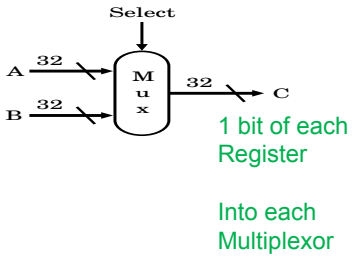


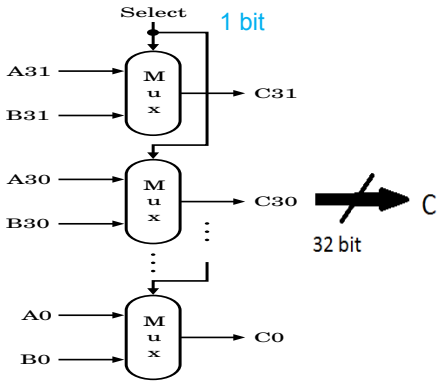
b. The 32-bit wide multiplex is actually an array of 32 1-bit multiplexors

Two Source Registers A and B

We would like to select only one

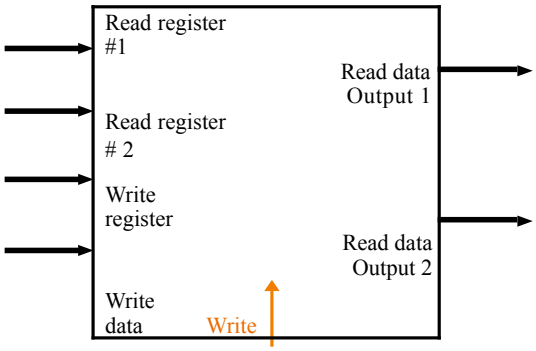
- Only one select bit needed that cascades into each multiplexor
- Output bits, $C_0 \ldots C_{31}$ combine together to form **C**





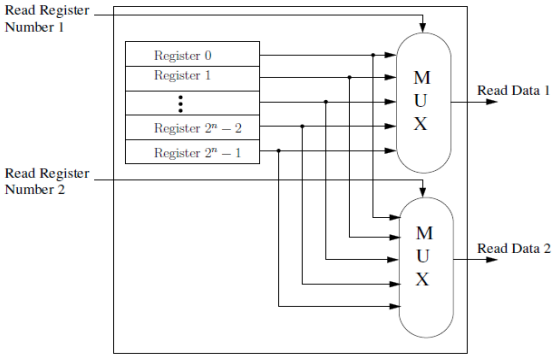
Reading from Register File

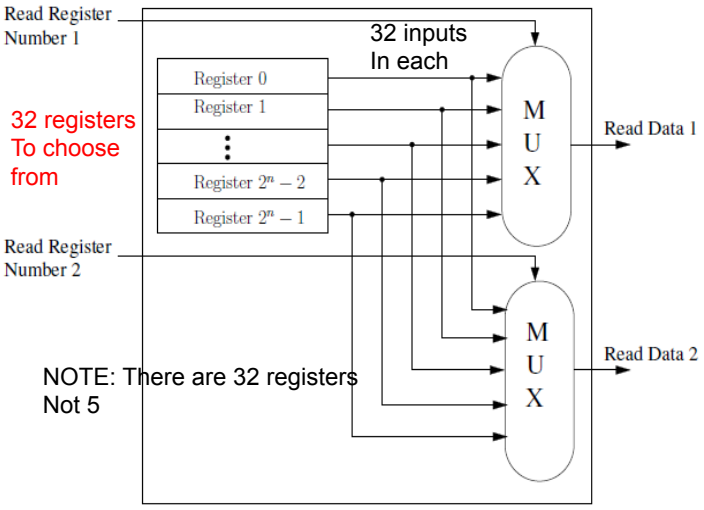
- Register: Also used for Reading from **two** registers at once



Reading from Register File

- Data from all 32 registers is sent out.
- Provide two numbers indicating which source registers are needed





32 registers To choose from

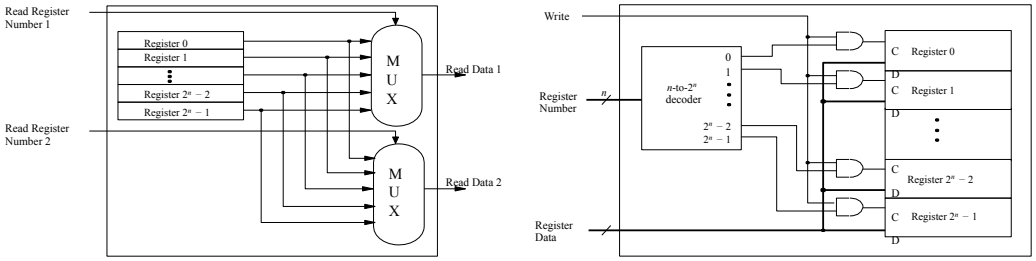
NOTE: There are 32 registers Not 5

Add instruction:
add $1, $2 , $3

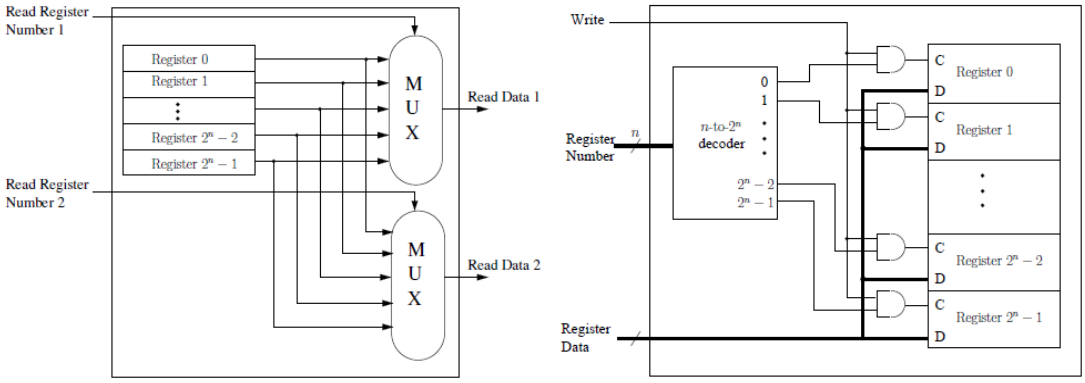
Read from two registers $2 and $3

Which register do we Read from: Depends on Selector lines

Read/Write Logic for Register File



Read/Write Logic for Register File



How many bits on Read Register Num1 and Num2?

How many bits, Write Register Data?

Using a Multiplexor to Do More:

- Use A Multiplexor to implement a Boolean Function
- Example Done in Class

Implementing Boolean Functions: ROMs

- Can think of ROM as table of 2^n m-bit words
- Can think of ROM as implementing m one-bit functions of n variables
- Internally, consists of a decoder plus an OR gate for each output
- Types of ROM: PROM, EPROM, EEPROM
- PLAs - simplified ROM
  Less hardware, but less flexible

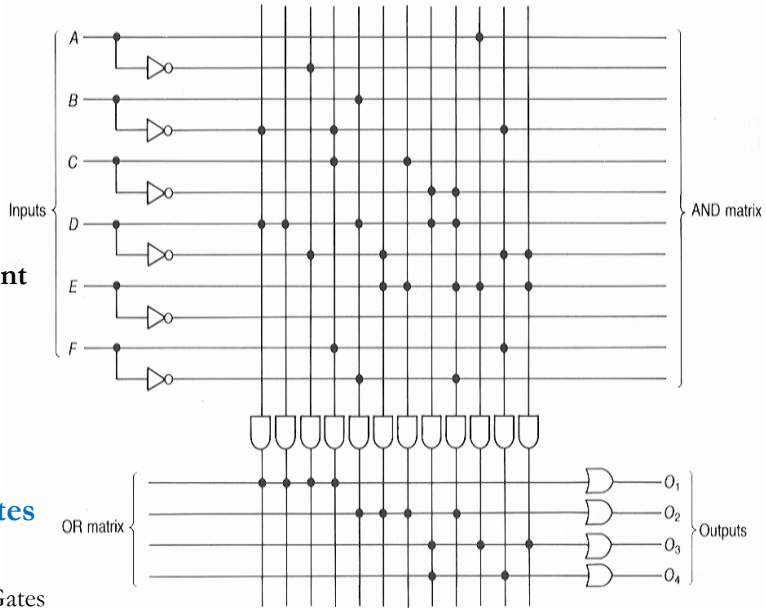
PLA/ ROM

We saw Truth Tables implement logic functions:

Functions of the form
*** Product of sums**

*** Layer of AND Gates followed by Layer of OR Gates**

- ROM/PLA:
- Internally AND Gates followed by OR Gates
- Every input and its complement are available



- **RAM (Random Access Memory)** this is memory that can be accessed very easily. The data is organized in such a way as to allow programs to access addresses efficiently. *Main memory*, easy access by CPU. Fast. Volatile storage. 'Save your work'
- **ROM (Read Only Memory)** is memory that cannot be changed. It can only be read –Audio CD is a good example of a ROM. Cannot be changed. Computer ROM retains its contents even when system is off. Non Volatile
- **RAM** usually can be written to and read from many times. **ROM** can only be written once, but it can be read many times.
- **Hard Drives:** Made up of several rigid metal, glass or ceramic disks. Moving parts, and slow access
- **SSD: Solid State Drives:** No moving parts to an SSD. Rather, information is stored in microchips. SSDs use Flash Memory- transistors stored in series such that information is retained even when power is off

Basic Digital Logic Design

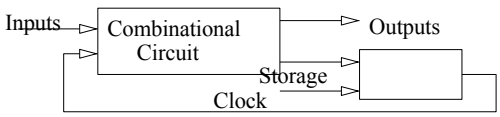
Sequential Circuits

Clocks and Sequential Circuits

- Two types of sequential circuits:
- Synchronous: has a clock
  Memory changes only at discrete points in time Clock pulse:

. . .

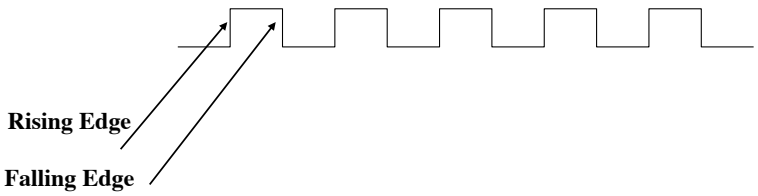
Block diagram:



  Easier to analyze, tend to be more stable
- Asynchronous: no clock
  Potentially faster and less power-hungry, but harder to design and analyze

Clocks and Sequential Circuits

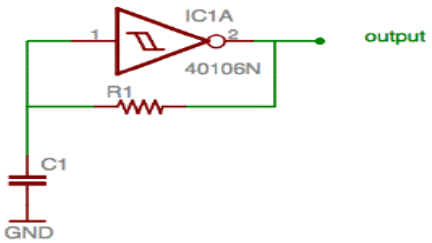
- Clock: Controlled input signal into the system



- Clock edge. Clock pulse ~ Clock cycle: From one rising edge to another rising edge

How Does a Clock Work: (FYI)
Many types of clock generators used in ICs today.

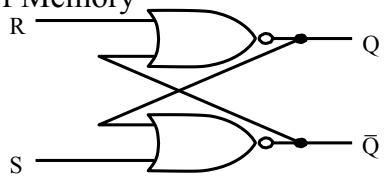
If you take an inverter and have the output drive the input, you get an oscillator. In the schematic below, the resistor/capacitor will control what frequency the circuit os



Basics: https://www.quora.com/How-exactly-is-a-clock-used-to-synchronize-parts-of-a-circuit

SR Latch with NOR gates

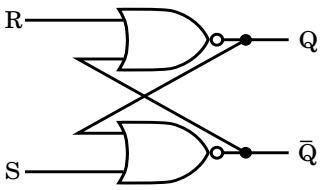
- Set/Reset Latch with NOR gates:
- Does this store 1 bit of Memory



- unclocked Memory bit: 1 bit recycles through the gates
- Steady state: S =R =0. System stays the way it is.
- S=1 (want to Set memory bit Q to 1)
- R=1 (want to Reset memory bit Q to 0)

Analyze SR Latch

- **When S=0, R=0** : Q is maintained
  - This holds true for Q=1 or Q=0

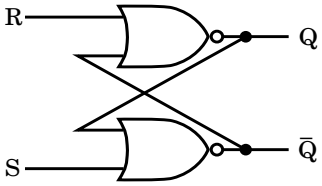


NOR GATE:

A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

Analyze SR Latch

- **When S=1, R=0** : Q is set to 1



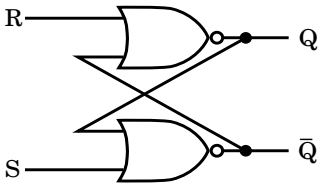
- Regardless of the previous value of Q

NOR GATE:

A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

Analyze SR Latch

- **When S=0, R=1** : Q is reset to 0



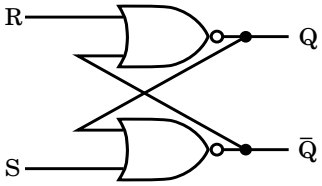
- Regardless of the previous value of Q
- If Q was previous 1 or 0 it is reset to 0

NOR GATE:

A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

Analyze SR Latch

- The System will return to steady state (S=0, R=0)



- One possibility not accounted for
- S=1, R=1 : What will occur

NOR GATE:

A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

S, R transition	$Q, \overline{Q}$ transition
$1,0 \rightarrow 0,0$	1,0 → 1,0
$0,1 \rightarrow 0,0$	0,1 → 0,1
$1,1 \rightarrow 0,0$	→

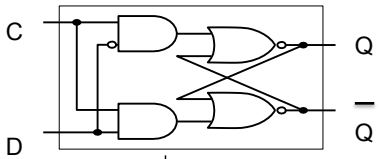
What happens here?

Functional Description of SR Latch

S	R	Q	$\overline{Q}$	
0	0	Q	$\overline{Q}$	Latch state (no change)
0	1	0	1	Reset state
1	0	1	0	Set state
1	1	?	?	Undefined

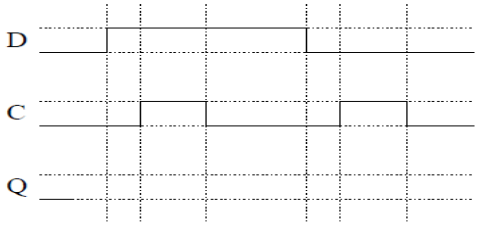
- Advantages:
  - Can "remember" value
  - Natural "reset" and "set" signals
    (SR=01 is "reset" to 0, SR=10 is "set" to 1)
- Disadvantages:
  - SR=11 input has to be avoided
  - No notion of a clock or change at discrete points in time yet

The D Latch

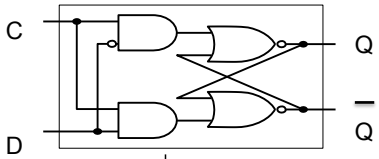


C	D	Next state of Q
0	X	No change
1	0	$Q = 0$ (Reset)
1	1	$Q = 1$ (Set)

Graphical example:

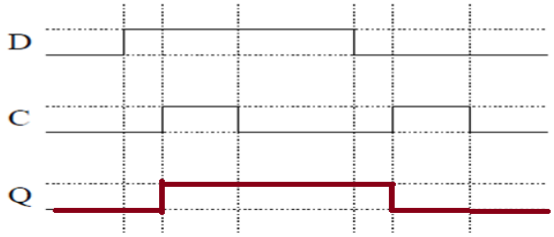


The D Latch



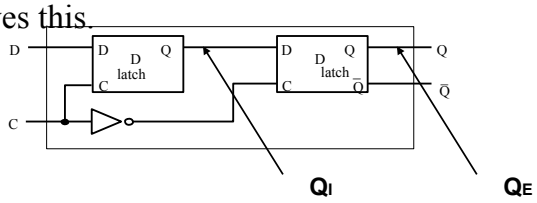
C	D	Next state of Q
0	X	No change
1	0	$Q = 0$ (Reset)
1	1	$Q = 1$ (Set)

Graphical example:



The D Flip-Flop

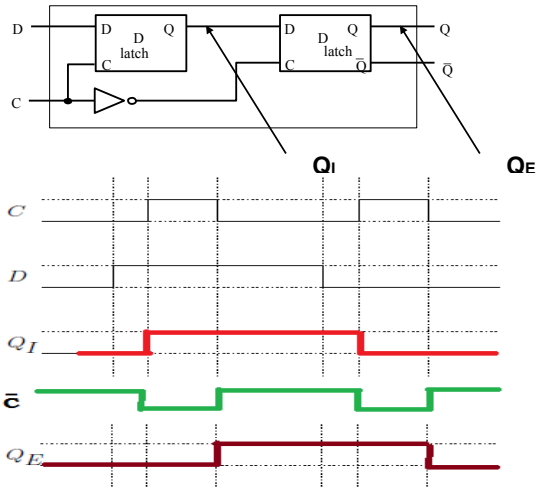
- We want state to be affected only at discrete points in time; a master-slave design achieves this.



- Clock pulse is inverted into second D-Latch

The D Flip-Flop

- Q_E related to original clock: Update occurs on falling edge of clock pulse



D-Latch, S-R Latch, D FLIP-FLOP : Which of the following IS TRUE :

A) S-R Latch is clocked

B) D-Latch uses a clock and controls the inputs to the S and R such that they can never both be 1.

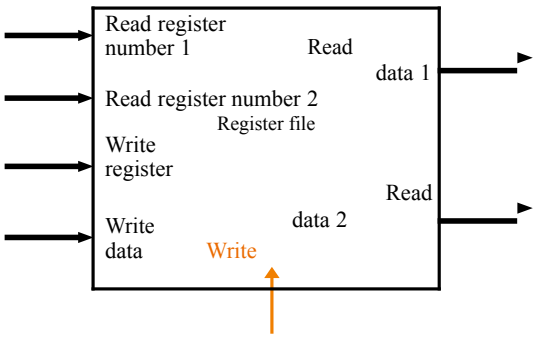
C) D Flip-Flop is able to store 2 bits

D) D Flip-Flop contains a series of 2 D-Latches and 4 S-R Latches

E) D Flip-Flop is useful only for large memories

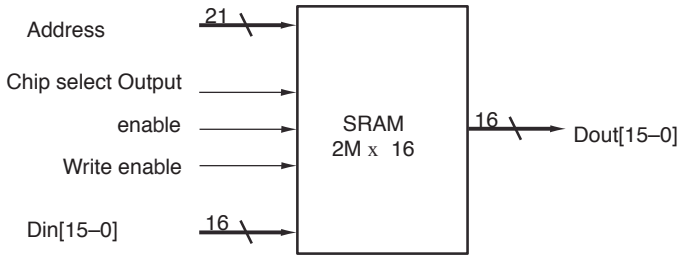
Registers and Register Files

- Each Register: One flip-flop per bit (e.g. 32 for a word register)



Random Access Memories

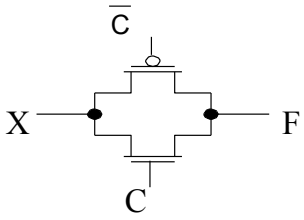
- Static random access memories (SRAM) use D latches



- Register file idea won't scale up; decoder and multiplexors too big
- Fix multiplexor problem by using three-state buffers
- Fix decoder problem by using two-level decoding
- This type of memory is **not** clocked

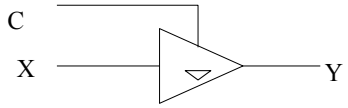
Three-state buffer or transmission gate

- Has three outputs 0, 1, and *floating* (connected to neither power or ground)

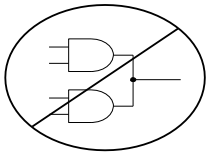


- $C = 1$, then
  - NMOS gate passes 0 well
  - $\overline{C} = 0$ and PMOS gate passes 1 well
- $C = 0$, then $\overline{C} = 1$ and both transistors are off (output is floating).

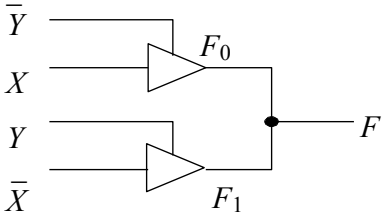
Using Three-State Buffers



- High-impedance outputs can be "tied together" without problems
- Normally, do not tie output lines together



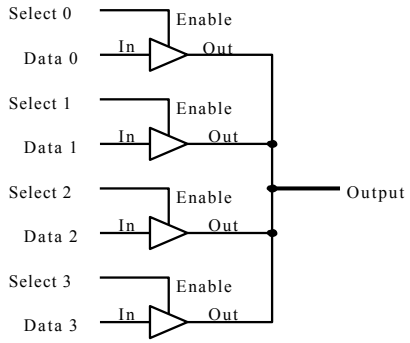
XOR from Three-State Buffers



Circuit analysis with transmission gates:

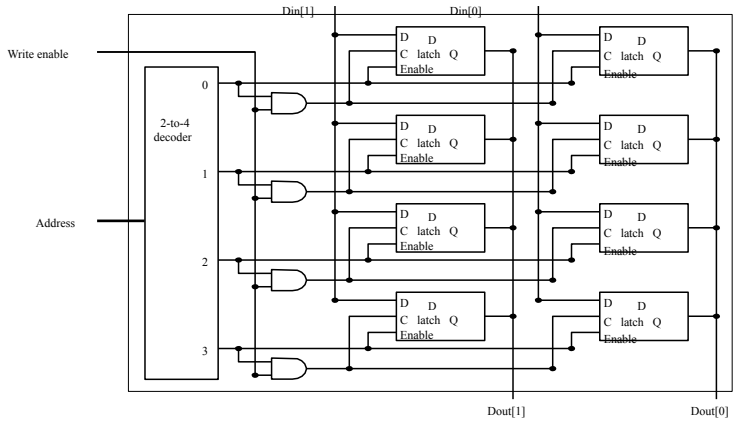
- Label floating output as '—'
- Tied lines better have exactly one non-floating!

Making Multiplexors from Three-State Buffers



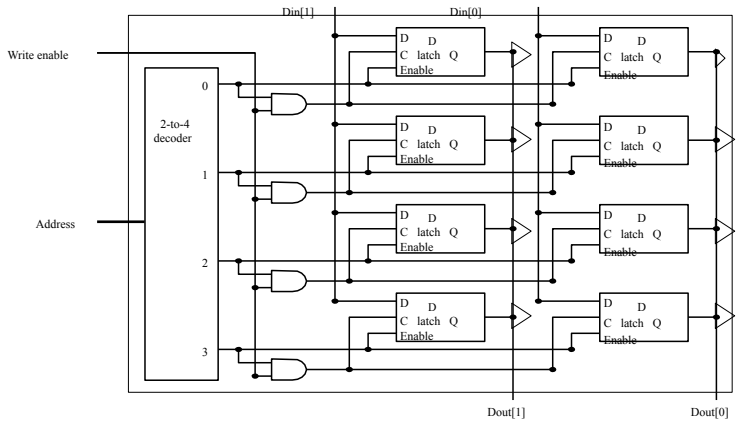
IMPORTANT: Must ensure that at most one select input is 1, or short-circuit may result (physical meltdown)

Example of SRAM Structure



Does this design scale up well?

Example of SRAM Structure

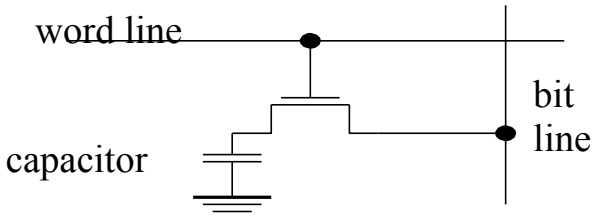


More accurately : tri-state buffers are needed in order to tie the output lines together

Dynamic RAM

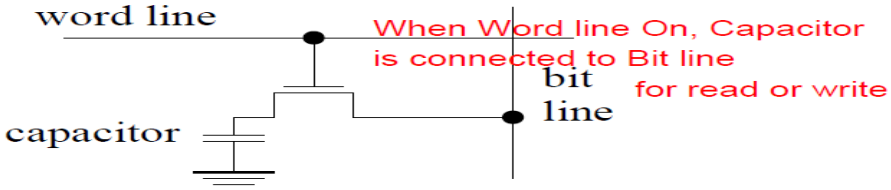
- Our SRAM cell uses a lot of transistors
- A better implementation uses six transistors
- This is still too expensive
- Alternative: use a capacitor to store a charge to represent 1
- Problem: charge leaks away, must be refreshed
- Capacitor is storing charge representing 0 or 1
- Unique about a capacitor- takes on the charge of whatever it is connected to. Charges and slowing the charge dissipates.

DRAM Cell



- To write: place value on bit line
- To read: put half-voltage on bit line, 1 on word line
- Charge in capacitor will slightly increase bit line voltage, no charge will slightly decrease voltage
- This is detected, amplified, and written back

DRAM Cell



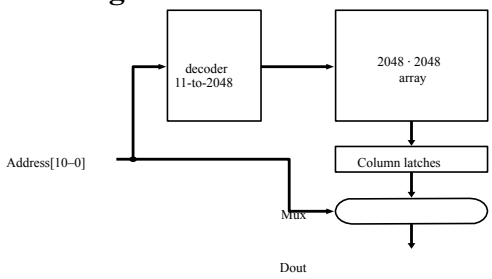
A Single DRAM Bit: One Capacitor One Transistor.

Cheaper But Slower

Transistor will allow us to write a charge or read a charge to the capacitor

Single Chip Memory Controller:
Dedicated to refreshing the capacitors within DRAM
Takes only 1-2% of active clock cycle time
Maximum 4%

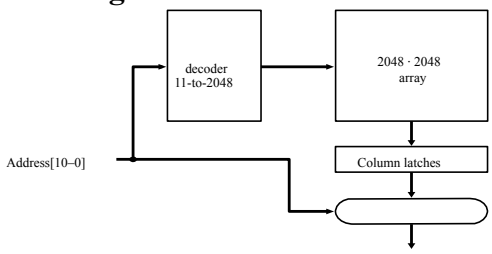
Design of 4Mx1 DRAM



Only 1 bit of data
In this case

- 22 bit address split up
- Whole row is read at once
- Column address selects single bit
- Refresh handled a row at a time (external controller)
- If capacitors hold charge for 4ms, refresh takes 80ns, fraction of time devoted to refresh is maximum 4%

Design of 4Mx1 DRAM



All the data from one row
Is sent through the
columns latches

1 bit of data

Two Level Decoding:

Address Bits : 11 go into the Decoder
11 bits go into the MUX to select the column
4M = Total 22 Bits

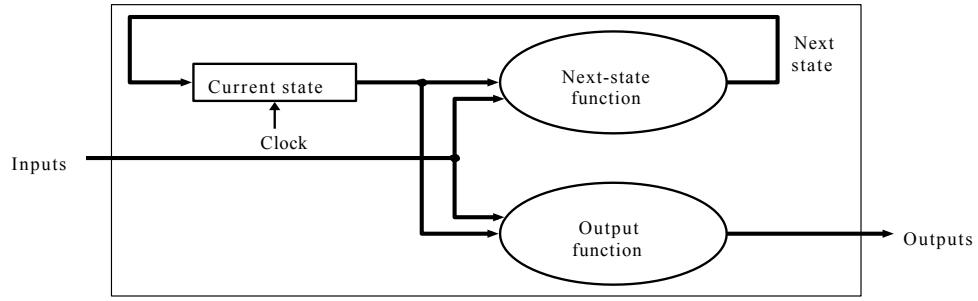
DRAM Complications

- DRAM is cheaper than SRAM, but slower
- Refresh controller must also allow read/write access
- Possibility of getting more bits out at a time (e.g. page-mode RAM)
- SDRAM: synchronized DRAM
  - Uses external clock to synchronize with processor
  - Useful in memory hierarchies

DRAM/ SRAM

SRAM: Expensive but Fast Cache
DRAM: Cheaper But Slower. Always need to refresh the Capacitors
Also a read from a capacitor will dissipate the charge
Read you need to read the charge and write it back

Designing Using Finite-State Machines



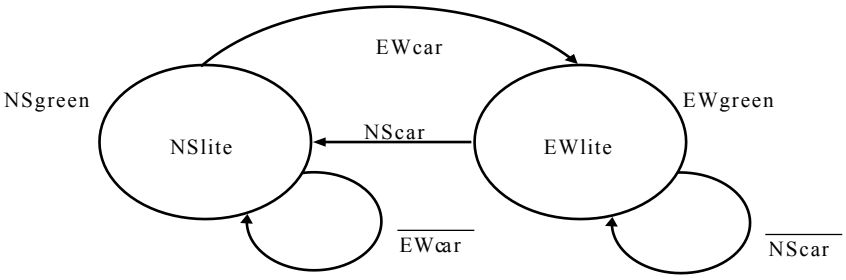
Graphical representation of a problem helps to define the behaviour of a system. Defining how inputs to the system will effect the state the system is in. In each state the system may or may not produce outputs.

FSM : Diagram
FSM : Truth Tables
FSM : Circuit

Example: Traffic Light

- Output signals: NSlight, EWlight
- Input signals: NScar, EWcar
- State names: NSgreen, EWgreen (no yellow for now)
- Functionality: want light to change only if car is waiting at red light

Graphical Representation of Traffic Light Controller

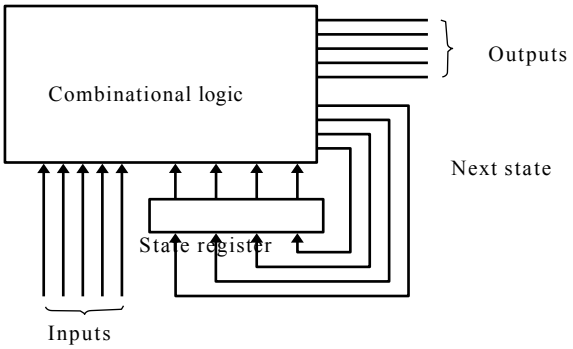


- Names of states outside ovals
- Output in given state inside oval
- Transition arc labelled with Boolean formula of inputs

Variations on Finite-State Machines

- Moore machine: output depends only on state (what we use)
- Mealy machine: output can depend on inputs
- Moore machine may be faster, Mealy machine may be smaller
- Conceptually, computation is infinite (input streams have no beginning or end)
- In practice, need to worry about power-up and power-down (as with all our state devices)
- Different in language-recognition context (e.g. CS 241)
  - Input is single character at a time, not set of bits
  - Because strings have finite length, computation is finite (start state, final states)
  - Mealy machines used (outputs on transition arcs)

Electronic Implementation of Finite-State Controller

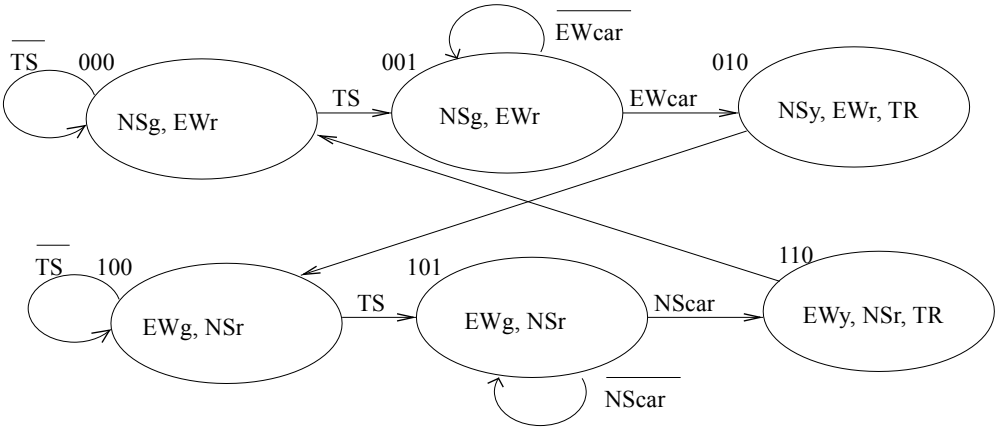


Extending the Traffic-Light Controller

- Add 4-second yellow light
- Assume 0.25Hz clock
- need to add 28-second timer
- Separate Controller for Timer
  - Input to timer: TimerReset (TR)
  - Output from timer: TimerSignal (TS)
- Behaviour of system
  - Stay green in one direction (red in other direction) until car arrives or 32 seconds elapse, whichever happens last
  - Green turns to yellow for 4 seconds; red in other direction stays
  - Yellow turns to red, red in other direction turns to green

State Diagram of Extended Controller

- Inputs: NScar, EWcar, TS
- Outputs: NSg, NSy, NSr, EWg, EWy, EWr, TR



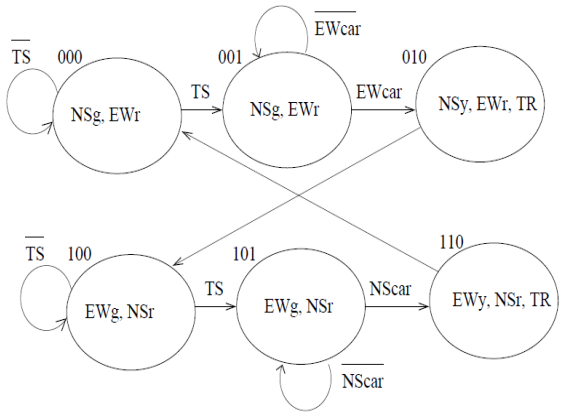
Next-State Table for Extended Controller

current state $S_2S_1S_0$	inputs NS-car	inputs EW-car	inputs TS	next state $S'_2S'_1S'_0$	current state $S_2S_1S_0$	inputs NS-car	inputs EW-car	inputs TS	next state $S'_2S'_1S'_0$
0 0 0	X	X	0	0 0 0	1 0 0	X	X	0	1 0 0
0 0 0	X	X	1	0 0 1	1 0 0	X	X	1	1 0 1
0 0 1	X	0	X	0 0 1	1 0 1	0	X	X	1 0 1
0 0 1	X	1	X	0 1 0	1 0 1	1	X	X	1 1 0
0 1 0	X	X	X	1 0 0	1 1 0	X	X	X	0 0 0
0 1 1	X	X	X	X X X	1 1 1	X	X	X	X X X

Note unused states, symmetries

State Diagram of Extended Controller

- Inputs: NScar, EWcar, TS
- Outputs: NSg, NSy, NSr, EWg, EWy, EWr, TR



Next-State Table for Extended Controller

current state $S_2S_1S_0$	inputs NS-car	inputs EW-car	inputs TS	next state $S'_2S'_1S'_0$	current state $S_2S_1S_0$	inputs NS-car	inputs EW-car	inputs TS	next state $S'_2S'_1S'_0$
0 0 0	X	X	0	0 0 0	1 0 0	X	X	0	1 0 0
0 0 0	X	X	1	0 0 1	1 0 0	X	X	1	1 0 1
0 0 1	X	0	X	0 0 1	1 0 1	0	X	X	1 0 1
0 0 1	X	1	X	0 1 0	1 0 1	1	X	X	1 1 0
0 1 0	X	X	X	1 0 0	1 1 0	X	X	X	0 0 0
0 1 1	X	X	X	X X X	1 1 1	X	X	X	X X X

Note unused states, symmetries

Next-State Table for Extended Controller

current state	inputs			next state	current state	inputs			next state
$S_2S_1S_0$	NS-car	EW-car	TS	$S_2'S_1'S_0'$	$S_2S_1S_0$	NS-car	EW-car	TS	$S_2'S_1'S_0'$
0 0 0	X	X	0	0 0 0	1 0 0	X	X	0	1 0 0
0 0 0	X	X	1	0 0 1	1 0 0	X	X	1	1 0 1
0 0 1	X	0	X	0 0 1	1 0 1	0	X	X	1 0 1
0 0 1	X	1	X	0 1 0	1 0 1	1	X	X	1 1 0
0 1 0	X	X	X	1 0 0	1 1 0	X	X	X	0 0 0
0 1 1	X	X	X	X X X	1 1 1	X	X	X	X X X

Current state = $S_2S_1S_0$, next state = $S_2'S_1'S_0'$

$$S_0' = \overline{S_1}\,\overline{S_0}\cdot TS + \overline{S_2}\,\overline{S_1}\,S_0\cdot\overline{EWcar} + S_2\overline{S_1}\,S_0\cdot\overline{NScar}$$

$$S_1' = \overline{S_2}\,\overline{S_1}\,S_0\cdot EWcar + S_2\overline{S_1}\,S_0\cdot NScar$$

Next State Functions: Determined by Inputs And Current State

Next-State/Output Logic For Extended Controller

Current state = $S_2S_1S_0$, next state = $S_2'S_1'S_0'$

$$S_0' = \overline{S_1}\,\overline{S_0}\cdot TS + \overline{S_2}\,\overline{S_1}\,S_0\cdot\overline{EWcar} + S_2\overline{S_1}\,S_0\cdot\overline{NScar}$$

$$S_1' = \overline{S_2}\,\overline{S_1}\,S_0\cdot EWcar + S_2\overline{S_1}\,S_0\cdot NScar$$

$$S_2' = \overline{S_2}\,S_1\overline{S_0} + S_2\overline{S_1}$$

Output Table For Extended Controller

- Output table looks like truth table
  Inputs are State, Outputs are Outputs
- Traffic light outputs: NSg, NSy, NSr, EWg, EWy, EWr, TR
- If output listed in State, then 1 in output table
  If output not listed in State, then 0 in output table

Output Table For Extended Controller

- Output table looks like truth table Inputs are State, Outputs are Outputs
- Traffic light outputs: NSg, NSy, NSr, EWg, EWy, EWr, TR
- If output listed in State, then 1 in output table
  If output not listed in State, then 0 in output table

S_2	S_1	S_0	NSg	NSy	NSr	EWg	EWy	EWr	TR
0	0	0	1	0	0	0	0	1	0
0	0	1	1	0	0	0	0	1	0
0	1	0	0	1	0	0	0	1	1
0	1	1	X	X	X	X	X	X	X
1	0	0	0	0	1	1	0	0	0
1	0	1	0	0	1	1	0	0	0
1	1	0	0	0	1	0	1	0	1
1	1	1	X	X	X	X	X	X	X

S_2	S_1	S_0	NSg	NSy	NSr	EWg	EWy	EWr	TR
0	0	0	1	0	0	0	0	1	0
0	0	1	1	0	0	0	0	1	0
0	1	0	0	1	0	0	0	1	1
0	1	1	X	X	X	X	X	X	X
1	0	0	0	0	1	1	0	0	0
1	0	1	0	0	1	1	0	0	0
1	1	0	0	0	1	0	1	0	1
1	1	1	X	X	X	X	X	X	X

OUTPUT FUNCTIONS

$$NSg = \overline{S_2}\,\overline{S_1},\ EWg = S_2\overline{S_1}$$

$$NSy = \overline{S_2}\,S_1\overline{S_0},\ EWy = S_2S_1\overline{S_0}$$

$$NSr = S_2,\ EWr = \overline{S_2}$$

S_2	S_1	S_0	NSg	NSy	NSr	EWg	EWy	EWr	TR
0	0	0	1	0	0	0	0	1	0
0	0	1	1	0	0	0	0	1	0
0	1	0	0	1	0	0	0	1	1
0	1	1	X	X	X	X	X	X	X
1	0	0	0	0	1	1	0	0	0
1	0	1	0	0	1	1	0	0	0
1	1	0	0	0	1	0	1	0	1
1	1	1	X	X	X	X	X	X	X

OUTPUT FUNCTIONS

$$NSg = \overline{S_2}\,\overline{S_1},\ EWg = S_2\overline{S_1}$$

$$NSy = \overline{S_2}\,S_1\overline{S_0},\ EWy = S_2S_1\overline{S_0}$$

$$NSr = S_2,\ EWr = \overline{S_2}$$

$$TR = S_1\overline{S_0}$$

Which is true about FSM

- A) we do not really need them in practice
- B) Specifies the behaviour of system where inputs **must** be used to transition from one state to another
- C) Specifies the behaviour of system where outputs depend on what position or state you are in the system.
- D) Both B and C
- E) None

Circuit: Implementation

- Circuit is based on the Tables derived from FSM diagram
- Next state is determined by Current state and Inputs

- Outputs in every state are determined
by only the current state

On Board

In Class FSM PROBLEM : Simplified Train Station

Given a Problem Definition: We derive the FSM

Description: Train on route between stations, picking and dropping passengers

***Inputs:** B (Buzz-button Pushed to Stop)
S (Train station is coming up yes or no)
P (Passengers waiting to enter the train)
***States:** The train can be in. What are the possibilities?
Think about this after reading specifications.
***Outputs:** **C** :Train Doors Closed(C=1)
Doors Closed (C=0)
R :Train is Running(R=1)
or not Running(R=0)

- Specifications:
- Train will keep running. It will receive a signal 'S' that a station is coming up.
- The Train will only stop at the station if there is also a B signal where a passenger pushed the buzzer **OR** a P signal indicating that passengers are waiting at the upcoming station.
- Assume S will always turn high before, B or P.
- If only S is true, the train will keep running
- Once the station is passed, S will turn low again
- If stopped at a station the Doors are open and the doors remain open as long as passengers keep entering.
- As soon as there are no more passengers- the train goes back to a Running state. Assume B, P reset.
- FSM Diagram On the black board

Data Representation and Manipulation

Data Representation and Manipulation

- Readings from text:
  - 2.4, 3.1, 3.2
  - 2.6
  - Appendix B (C in 4th Edition)
- How characters and numbers are represented in a typical computer
- Hardware designs which implement arithmetic operations

The MIPS Word

- 32-bit architecture
- 1 byte = 8 bits; 4 bytes = 1 word
- Bits numbered 31, 30, …, 0
- Most significant bit (MSB) is bit 31
- Least significant bit (LSB) is bit 0
- In many examples, we will use only 4 bits to illustrate
- Sometimes, numbers written in *hexadecimal*

Characters

- ASCII (American Standard Code for Information Interchange)
- Uses 7 bits to represent 128 different characters
- 8th bit (topmost) used as parity check (error detection)
- 4 characters fit into MIPS 32-bit word
  128 possibilities include upper and lower case Roman letters, punctuation marks, some computer control characters
- Partial table on page 106 of text
- Unicode: 16 bits per character (English isn't the only language!)

USASCII code chart

b_7 b_6 b_5 → / b_4 b_3 b_2 b_1 ↓ Bits / Row ↓ Column →	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
	0	1	2	3	4	5	6	7
0 0 0 0	0	NUL	DLE	SP	0	@	P	`	p
0 0 0 1	1	SOH	DC1	!	1	A	Q	a	q
0 0 1 0	2	STX	DC2	"	2	B	R	b	r
0 0 1 1	3	ETX	DC3	#	3	C	S	c	s
0 1 0 0	4	EOT	DC4	$	4	D	T	d	t
0 1 0 1	5	ENQ	NAK	%	5	E	U	e	u
0 1 1 0	6	ACK	SYN	&	6	F	V	f	v
0 1 1 1	7	BEL	ETB	'	7	G	W	g	w
1 0 0 0	8	BS	CAN	(	8	H	X	h	x
1 0 0 1	9	HT	EM	)	9	I	Y	i	y
1 0 1 0	10	LF	SUB	*	:	J	Z	j	z
1 0 1 1	11	VT	ESC	+	;	K	[	k	{
1 1 0 0	12	FF	FS	,	<	L	\	l	|
1 1 0 1	13	CR	GS	-	=	M	]	m	}
1 1 1 0	14	SO	RS	.	>	N	^	n	~
1 1 1 1	15	SI	US	/	?	O	_	o	DEL

128 ASCII characters: 1972

Basic Error detection when transmitting Data

- Parity Bit: Top most bit, very basic error detection.
- When bits, bytes(8bits), words(32 bits) are transmitted from computer to computer, parity check in case any error in transmission
- 10011011 : Highest bit states if there is an even or odd number of 1's in the 7 bit number: 1 indicates there is an even number of 1s
- Could also be the 9th bit in an 8bit number: 000110111
- Even parity: highest order bit is true when there are even number of 1s
- Odd parity: highest order bit is true when there are an odd number of 1s

Unsigned Binary Numbers

- With 4 bits, can represent 0 through 15

$$1101_2 = (1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0)$$
$$= 13_{10}$$

- With 32 bits, can represent 0 through $2^{32} - 1 = 4,294,967,295$
- How to represent negative numbers?

Basic Binary Addition/Subtraction:

Addition:

```
1010  :10
0011  :3
--------
1101  :13
```

Subtraction:

```
1010  :10
0011  :3
--------
0111  :7
```

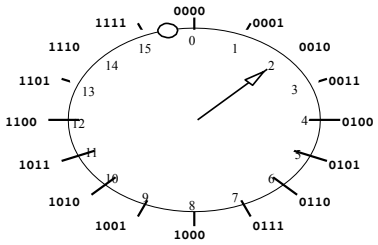
Signed Binary Numbers

- First idea: use MSB as "sign bit" (0 means positive, 1 means negative)
- Called "signed-magnitude" representation
- 4-bit example: 1110 is − 6
- With 4 bits, can represent − 7 (1111) to + 7 (0111)
- Problems: two different versions of zero (0000 and 1000), addition is complicated
- Better idea: two's complement representation

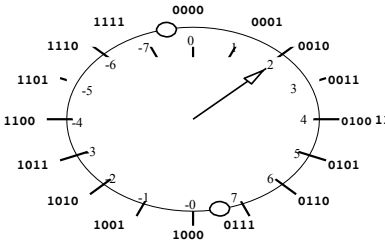
Two's Complement Representation

- Idea: Let MSB represent the negative of a power of 2
- With 4 bits, bit 3 (MSB) represents -2^3
- $1110 = -2^3 + 2^2 + 2^1 = -2$
- With 4 bits, can represent − 8 (1000) to + 7 (0111)
- With 32 bits, can represent − 2, 147, 483, 648 to 2,147,483,647
- Usefulness becomes apparent when we try arithmetic

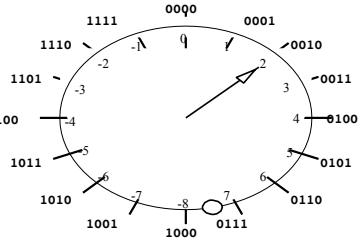
Pictorial Representation, 4 Bits



Unsigned



Signed Magnitude



Two's Complement

Negating a Two's Complement Number

- For a bit pattern x, let $\bar{x}$ be the result of inverting each bit
- Example: $x = 0110$, $\bar{x} = 1001$
- Since $x + \bar{x} = -1$, $-x = \bar{x} + 1$
- To negate a number in two's complement representation, invert every bit and add 1 to the result

0110 :+6
1001 :negate all bit
0001 :add 1

1010 :-6 in Two's Complement

Sign Extension

- With 4 bits, 0110 is +6. With 8 bits, what is +6?
- With 4 bits, 1010 is − 6. With 8 bits, what is − 6?
- To expand number of bits used, copy old MSB into new bit positions.
- This works because

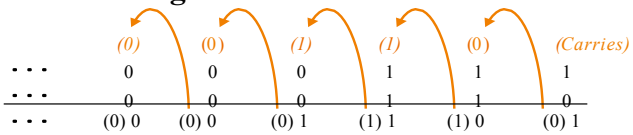
$$-2^i + 2^{i-1} + 2^{i-2} + \cdots + 2^{j+1} + 2 \cdot 2^j = 0$$

  - For example : -4 1100 in 8 bits. Simply extend MSB:
  - ➔ 1111 1100 = -4 in two's complement form

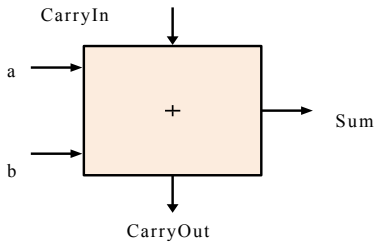
Addition

- To add two two's complement numbers, simply use the "elementary school algorithm", throwing away any carry out of the MSB position
- To subtract, simply negate and add
- Problem: what if answer cannot be represented? (called overflow)
- Overflow in addition cannot occur if one number is positive and the other negative
- If both addends have same sign but answer has different sign, overflow has occurred

Building An Addition Circuit

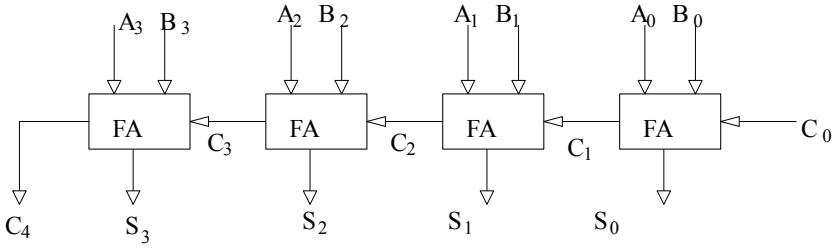


Basic building block: Full Adder
(takes three bits as input, outputs 2-bit sum)



Ripple-Carry Adder

- 4-bit example



- Easy to extend to 32 bits
- Can be slow; “carry-lookahead” idea improves speed

Faster Carry Or Look Ahead Adder:

- In a row of adders: try to generate what is the next carry into column on left
- All the bits of numbers A and B are known at beginning.
- Carry-In to each adder is unknown.
- This is beyond the scope of this course
- But in the notes FYI

Faster Carry Or Look Ahead Adder:

- c1 = (a0·c0) + (b0·c0) + (a0·b0)
- c2 = (a1·c1) + (b1·c1) + (a1·b1)
- **c2 = (a1·a0·b0) + (a1·a0·c0)·(a1·b0·c0) + (b1·a0·b0)+ (b1·a0·c0)+ (b1·b0·c0)+ (b1·a1)**

can predict the next carry based on a, b values and c0.

Predict Carry

c2 = (a1·a0·b0) + (a1·a0·c0)·(a1·b0·c0) + (b1·a0·b0)+(b1·a0·c0)+ (b1·b0·c0)+ (b1·a1)

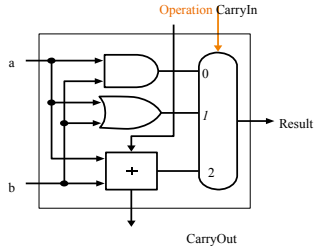
In this manner we can express the carry-in at every level

- Therefore, additional gates, added into each adder.
- Next level, gets more complicated. Simple substitution into the same formula.
- Hardware quickly adds up and can get expensive for each bit
- Carry look ahead adder, much more efficient and some level of look ahead is added into each adding unit.
- **Fast carry using propagate and generate : Read about it in the Appendix**

Logical Operations

- shifting bits in word left or right (shifting in 0, called "logical shift")
- shift right, duplicating old MSB (called "arithmetic right shift")
- rotate left or right (moving bit rotated out to other side)
- bitwise AND, OR, NOR (bitwise NOT is just NOR with all-zero word)

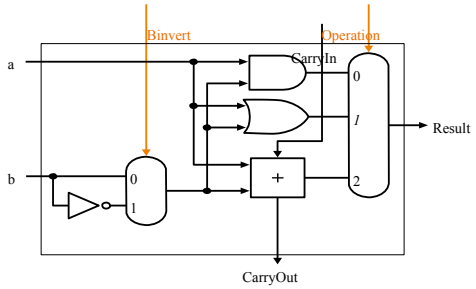
A 1-Bit ALU



- Extends functionality of full adder
- Performs AND, OR, addition
- Connect 32 of these as with ripple-carry adder to perform 32-bit operations

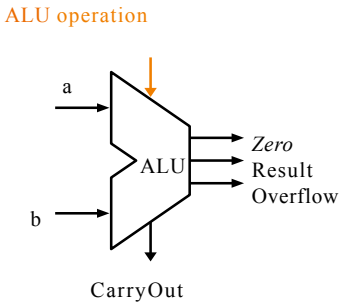
Improving the 1-Bit ALU

- How to implement subtraction?
- To subtract *b* from *a*, invert bits of *b*, add to *a*, add 1
- Box below will do this, if added 1 is put into CarryIn at top of chain when subtraction is desired



Abstracting Away ALU Details

- Book makes further improvements to support other operations that assist in branching (Appendix B)
- From now on, we use symbol below
- Same shape used for ripple-carry adder, so remember to label them



Multiplication

- Example:

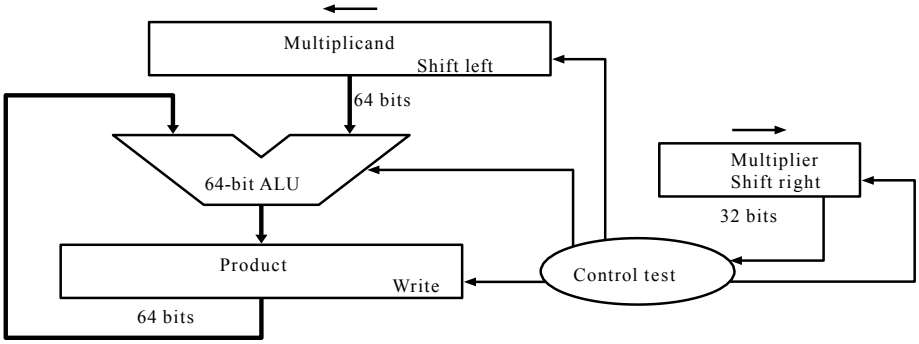
13					1	1	0	1	Multiplicand
11					1	0	1	1	Multiplier
					1	1	0	1	
				1	1	0	1		
			0	0	0	0			
		1	1		0	1			
143	1	0	0		0	1	1	1	1	Product

- Algorithm for Product = Multiplicand * Multiplier:
- Look at LSB of Multiplier

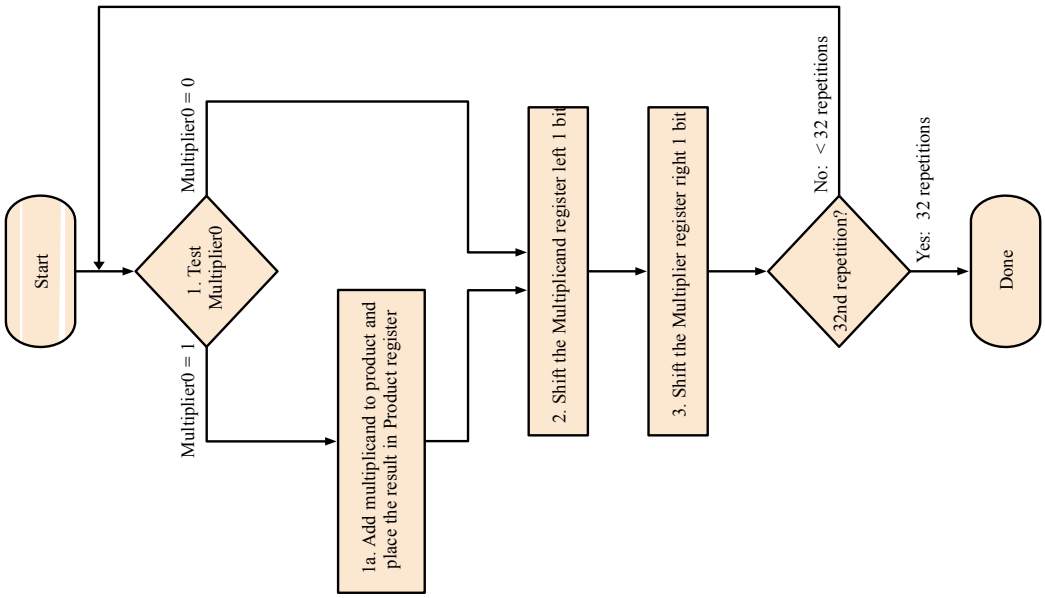
  If 1, add Multiplicand to Product

  Shift Multiplicand left, Shift Multiplier right Repeat until Multiplier becomes zero
- Note: for *n* bit numbers, result may be 2*n* bits

Multiplication Hardware, First Version



- Initialization and termination not shown
- At start, Product is zero, 32-bit Multiplicand in right half of its register
- Note control inputs and outputs

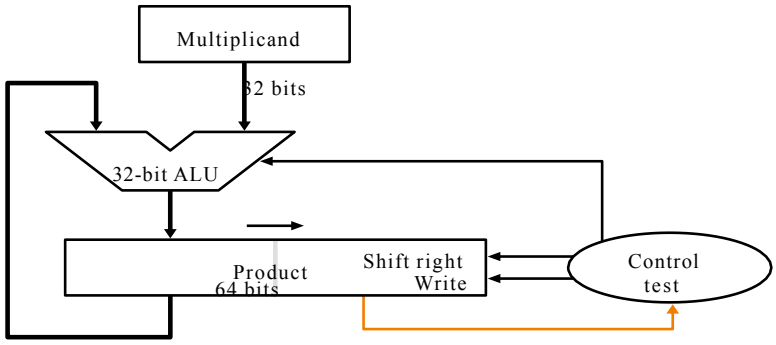


4-Bit Multiplication Example, First HW Version

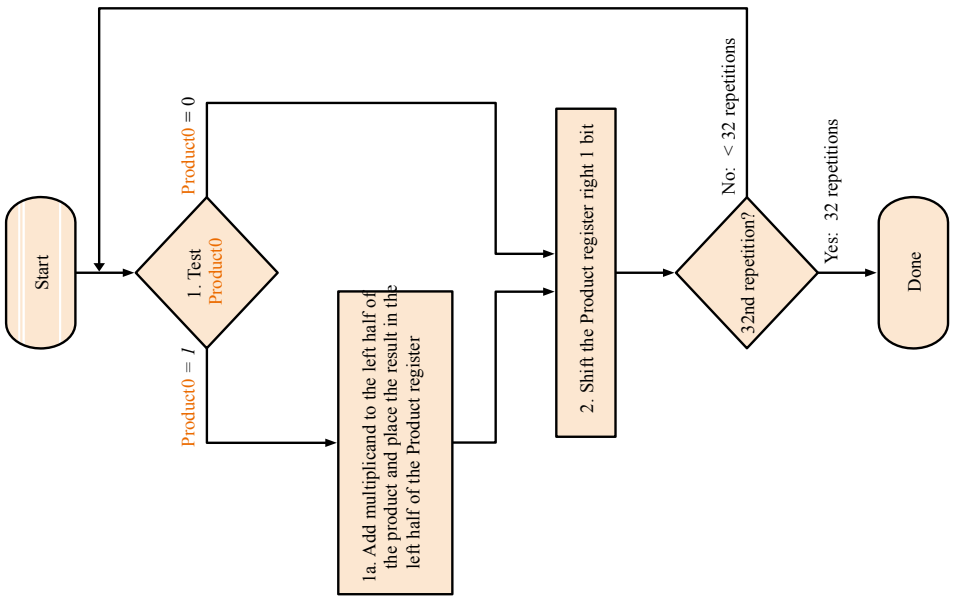
Multiplier = 1011, Multiplicand = 1101

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial Values	1011	0000 1101	0000 0000
1	Add mpcd to prod Shift left mpcd Shift right mplr	0101	0001 1010	0000 1101
2	Add mpcd to prod Shift left mpcd Shift right mplr	0010	0011 0100	0010 0111
3	No operation Shift left mpcd Shift right mplr	0001	0110 1000	
4	Add mpcd to prod Shift left mpcd Shift right mplr	0000	1101 0000	1000 1111

Multiplication Hardware, Third Version



- Multiplier starts in right half of product register
- As multiplier bits are shifted out, unchanging bits of product are shifted into the space created
- Optimizations:
  - Smaller storage registers , Smaller ALU , Eliminate need for Multiplier separate register

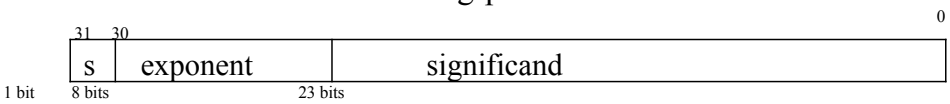


Representing Numbers That Aren't Integers

- Uses idea of scientific notation: -3.45×10^3
- Sign, significand (fraction, mantissa, exponent)
- Normalized: single digit to left of decimal point
- For computers, natural to use 2 as base
- Example: $1.01_2 \times 2^4$
- In normalized binary, leading digit of significand is always 1 (can omit it from internal representation)
- How to represent 0?

Floating-Point Representation

- MIPS uses the IEEE 754 floating-point standard format



- allows numbers from 2.0×10^{-38} to 2.0×10^{38}, roughly
- Double precision: uses two 32-bit words, 11 bits for exponent, 52 bits for significand
- Exponent is stored in "biased" notation: most negative exponent is all 0's, most positive is all 1's
  This allows for quick comparisons, speeds up sorting
- Thus value represented is
  $(-1)^{S} \times (1 + \text{Significand}) \times 2^{(\text{Exponent} - \text{Bias})}$, where Bias = 127 for single precision
- Special case: 0000 0000 exponent reserved for 0

Bias notation : IEEE Floating Point standard

- Let 1111 1111 be the most positive exponent
- 0000 0000 be the most negative exponent.
  - Makes sorting easier
- Positive exponent: takes positive binary number represented by exponent bits and subtract 127 from this.
- Therefore all exponents have positive value: take the value minus the bias
- Exponent of +1: 1000 0000 **(128 -127)**
- Exponent of -1: 0111 1110 **(126-127)**

IEEE 754 Standard: Floating Point Representation

- The representation is meant to make sorting and comparing numbers easier
  - This is why sign is most significant bit
  - Also why exponent bits are before significand bits
- Based on *COAD Text (H&P)* representing 0:
  - 0000 0000 RESERVED for **Zero FP number**
  - 1111 1111 RESERVED for exceptions beyond scope of normal floating point numbers. Such as de-normalized numbers

  Therefore lowest exponent is -126, highest positive exponent +127

Fractional Numbers

- How to represent numbers less than 1?
- Digits to right of decimal point represent negative powers of two

0.	1	0	1	1
1	1/2	1/4	1/8	1/16

0*1 + 1*1/2 + 0*1/4 + 1*1/8 + 1*1/16 = 11/16

- Simple examples 1/2 = 0.1

  3/4 = 0.11
- May have to approximate Example: 1/3 as decimal is...

  0.1 in binary is... sqrt(2) in binary is...

Algorithm for conversion of fractions

- Multiply fraction by 2 repeatedly.
- 0.625 x 2 = **1**.25 KEEP 1 as first binary digit **0.1**
- Next .25 x 2 = **0**.5 : 0 as next binary digit : **0.10**
- Next .5 x 2 = **1**.0 : 1 as next binary digit : **0.101**
- **Done**
- **.625 is .101 as binary exactly**
- **NOT ALL fractions can be represented in binary exactly**

Algorithm for conversion of fractions

- 0.1 decimal 1/10
- 0.1 x 2 = **0.2** KEEP 0 as first binary digit **0.0**
- 0.2 x 2 = **0.4** KEEP 0 as next binary digit **0.00**
- 0.4 x 2 = **0.8** KEEP 0 as next binary digit **0.000**
- 0.8 x 2 = **1.6** KEEP 1 as next binary digit **0.0001**
- 0.6 x 2 = **1.2** KEEP 1 as next binary digit **0.00011**
- **Repeat with 0.2 will lead to**
- 0.1 x 2 = **0.2** KEEP 0 as first binary digit **0.0 …. repeating**
- **KEEP Going .00011000110001100011 . . .**
- **Repeating pattern**
- **Therefore some numbers produce infinite binary expansion.**

Convert this Fractional number to IEEE Floating Point Representation

Apply the same algorithm from previous slides

Start: Convert Decimal 2.625
2 = 10

.625 is .101 as binary

➔2.625 is 10.101 as binary

Need to Normalize : Only one leading 1

1.0101 x 2^1
SIGN BIT: 0 (positive)
Exponent: x – 127 = 1, Therefore x = 128

As 8 Bits: 1000 000
Significand Bits Or Precision Bits: 0101
FINAL IEEE Representation:
➔ 0 1000 0000 0101 0000 0000 0000 0000 000

Floating-Point Addition

- Decimal example: $9.54 \times 10^2 + 6.83 \times 10^1$
  (assume we can only store two digits to right of decimal point)
  1. Match exponents: $9.54 \times 10^2 + .683 \times 10^2$
  2. Add significands, with sign: 10.223×10^2.
  3. Normalize: 1.0223×10^3
  4. Check for exponent overflow/underflow
  5. Round: 1.02×10^3
  6. May have to normalize again
- Same idea works for binary

Floating-Point Addition

A: 0 1000 0011 0001000000 : Exp 4
B: 1 1000 0100 1110000000 : Exp 5

Must Shift A's Mantissa in order to match the exponents

A: 1.000100 x 2^4
B: 1.111000 2^5
Becomes: A: 0.10001000 x 2^5
B: 1.11100000x 2^5

Note B is a negative number, B is larger

Subtract: 1.11100000
0.10001000

1.01011 x 2^5

Note, I may have to Renormalize here

Floating-Point Addition

A: 0 1000 0011 0001000000 : Exp 4
B: 1 1000 0100 1110000000 : Exp 5

Subtract: 1.11100000 x 2^5
0.10001000 x 2^5

1.01011 x 2^5

Note, I may have to Renormalize here and therefore may need to update exponent value also (not in this case)

Final Answer:
Negative: B was larger:
Exponent: 5.
Mantissa: 01011 (first 1 is implicit)

1 1000 0100 01011 0000 …

Floating-Point Multiplication

- Decimal example: $(9.54 \times 10^2) \times (6.83 \times 10^1)$
  (assume we can only store two digits to right of decimal point)
  1. Add exponents: $2 + 1 = 3$
     (Note: exponents stored in biased notation)
  2. Multiply significands: $9.54 \times 6.83 = 65.1582$
  3. Unnormalized result: 65.1582×10^3
  4. Normalize: 6.51582×10^4
  5. Check for overflow/underflow
  6. Round: 6.52×10^4
     (May need to renormalize)
  7. Set sign
- We would do it the same way in Binary using IEEE representation

Single Cycle Computer

Datapaths

Single-Cycle Processor Implementation

- Readings from text: Chapter 4, sections 4.1–4.4; Appendix D, section D.2
- How to build datapath, control for specific architecture
- We will implement small subset of MIPS operations:
  - Load (lw) and store (sw)
  - Add (add), subtract (sub), AND (and), OR (or), set on less than (slt)
  - Branch-equal (beq) and jump (j)
- These suffice to illustrate fundamental ideas

Review of MIPS Architecture

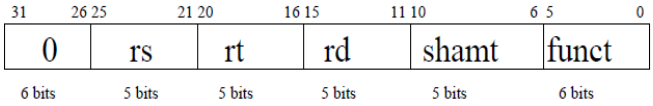
- 32 registers (numbered 0 to 31), each with 32 bits
- Register 0 always supplies the value 0
- Memory of 32-bit words
- Memory is byte-addressable (word addresses multiples of 4)
- Words have the address of their least significant byte
- All MIPS instructions are 32 bits long

Review of MIPS Instructions

- Load: lw $s1, 100($s2)
  - Operands are register to be loaded, address in memory
  - Addressing modes: register, base (displacement), immediate
- Add: add $s1, $s2, $s3
  - Operands are destination register, two source registers
- Branch equal: beq $s0, $s1, 10
  - Operands are registers to be compared, relative jump offset
- Jump: j 3000
  - Operand is word address of next instruction (need to multiply by 4)

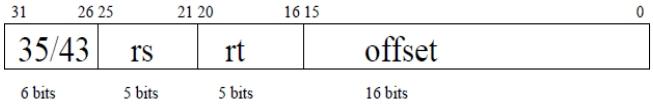
How the Instruction Bits are Broken up:

R-format: `add $t1, $t2, $t3` ⇒ `add rd,rs,rt`



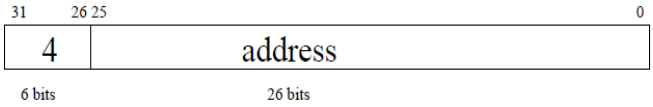
sll, sub

Load/store: `lw $t1, 100($t2)` ⇒ `lw rt,100(rs)`



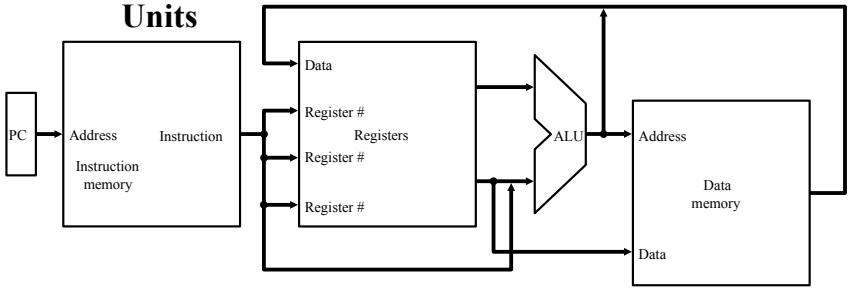
beq, subi, addi

Jump: `j 3000`



Special: Only jump instruction

High-Level View of MIPS Functional Units

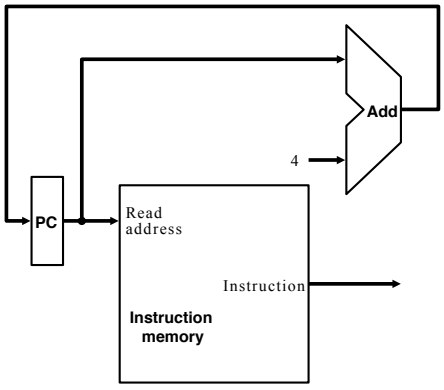


- PC: Program Counter (address of current instruction)
- Fetch-execute cycle:
  - Fetch instruction (update PC)
  - Execute instruction
    - Fetch register operands
    - Compute result
    - Store into registers OR use to index memory

First Implementation: One Cycle Per Instruction

- Simpler to understand, but not practical
- Requires separate instruction and data memories
- Clock must be slowed to speed of slowest instruction
- Subsequently we look at multicycle implementations

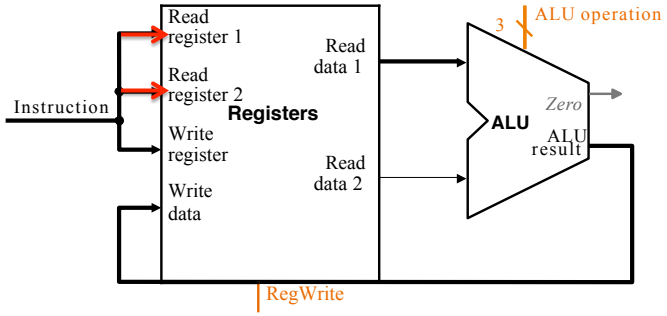
Implementing Fetch Portion of Fetch-Execute



- State elements here are PC (register) and instruction memory
- Adder is combinational

Datapath components for R-type instructions

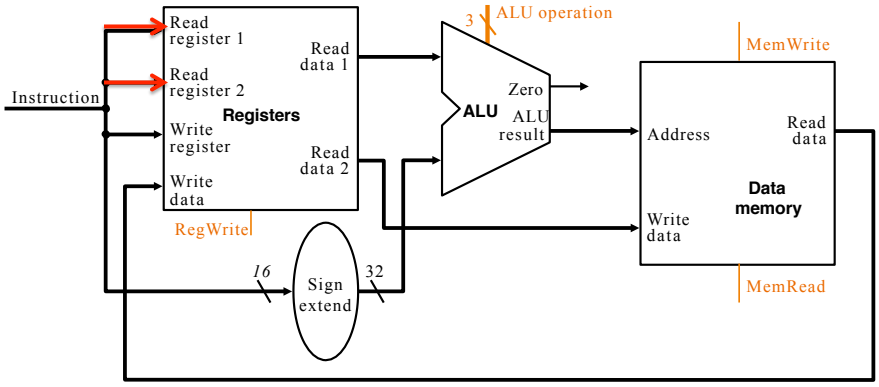
- Example: add $t1, $t2, $t3



- Note design permits read/write of same register

Datapath components for load/store instructions

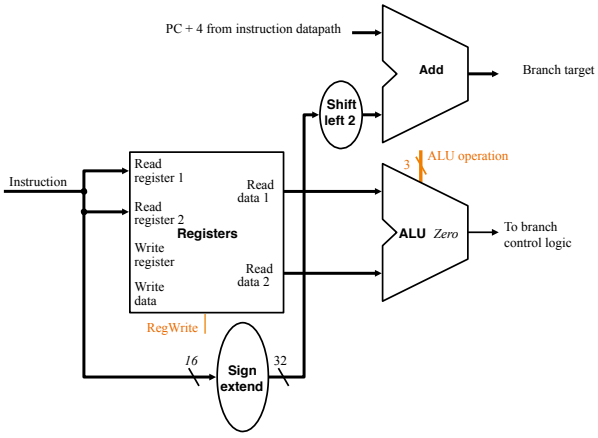
- Example: lw $t1, 100($t2)



- Sign extend is combinational
- Assume for simplicity that data memory is edge-triggered

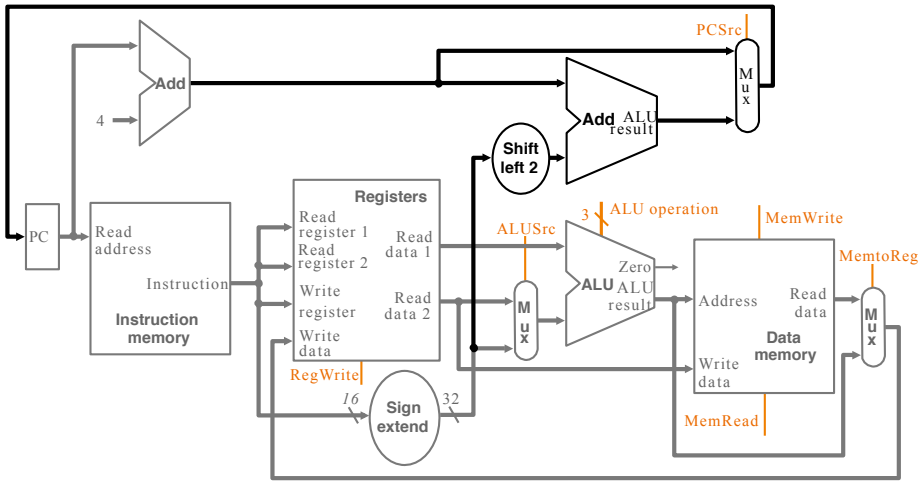
Datapath for Branch Instruction

- Example: beq $t1, $t2, 100



- Shift is necessary because offset given is in words
- Still need mechanism to control PC loading

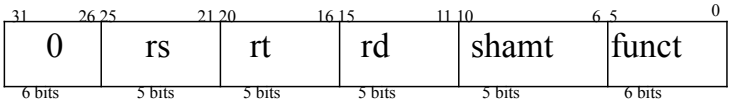
Assembled Single-Cycle Datapath



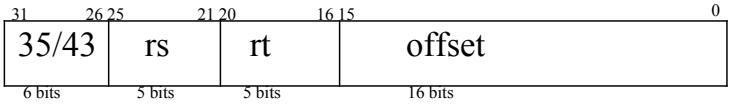
- Note: No control unit yet
- Multiplexors added to “reuse” units
- Let’s examine an R-Format instruction : add $t1, $t2, $t3

Instruction Formats

R-format: add $t1, $t2, $t3



Load/store: lw $t1, 100($t2) ⇒ lw rt,100(rs)

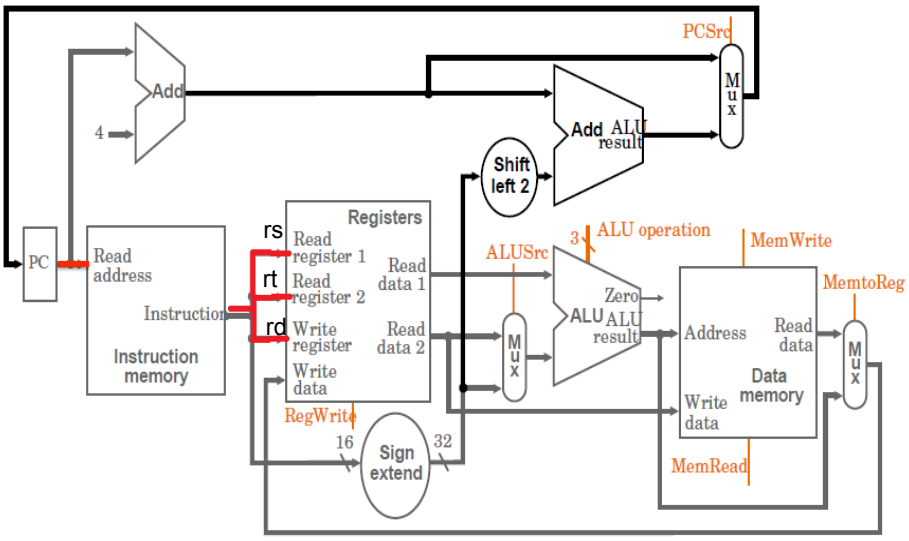


Jump: j 3000

- First field is operation code (opcode) but add, sub, etc. don't need separate opcodes (funct)
- Note "destination register" field is different for add and lw; this complicates the datapath some more

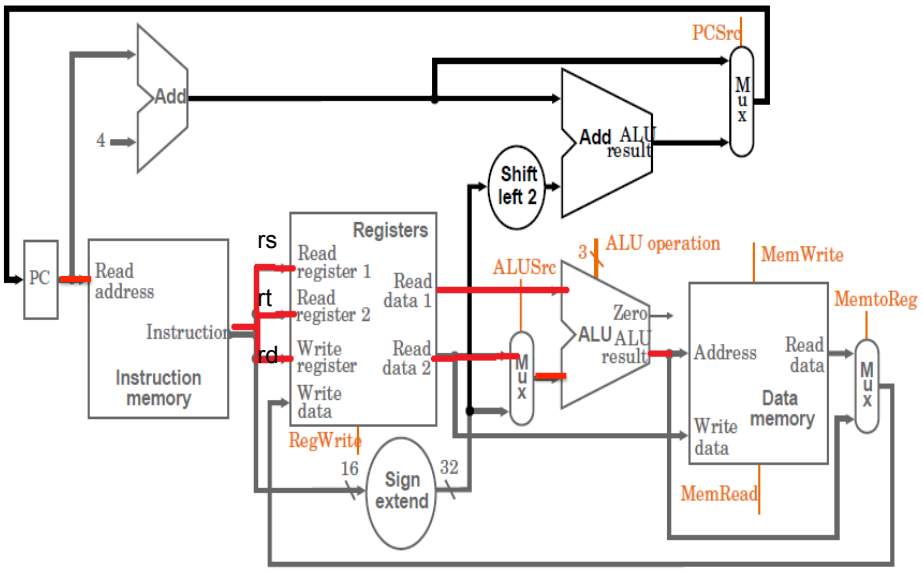
What Statement is False about the DATAPATH:

A. Data Memory is separate from Instruction Memory
B. Data Mem and Instruction Mem are a type of Cache
C. Register file is accessed for 2 registers in parallel
D. ALU can be used for multiple source operands.
E. Both memories can be read from and written to in the datapath

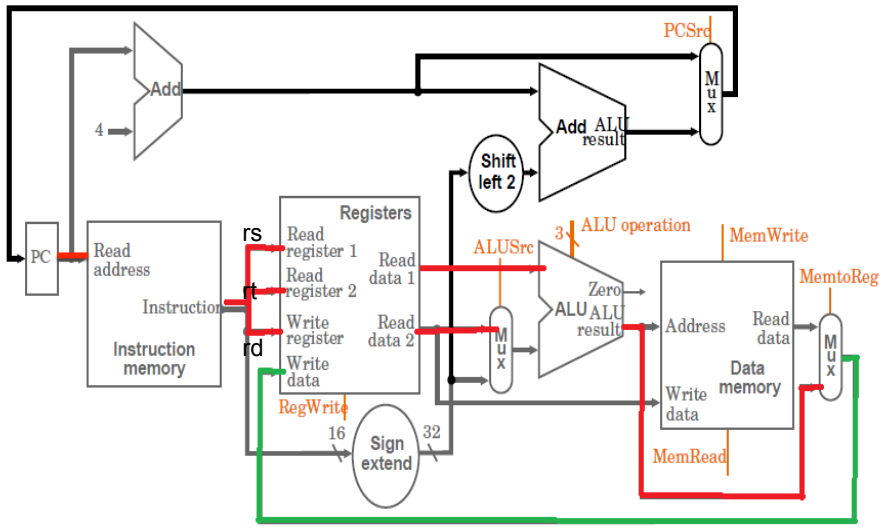


add $t1, $t2, $t3

First Step: Always Fetch Instruction from Instruction Memory(use PC)
Next: Use all the Instruction Bits
***Access the Register File with the two source register numbers**

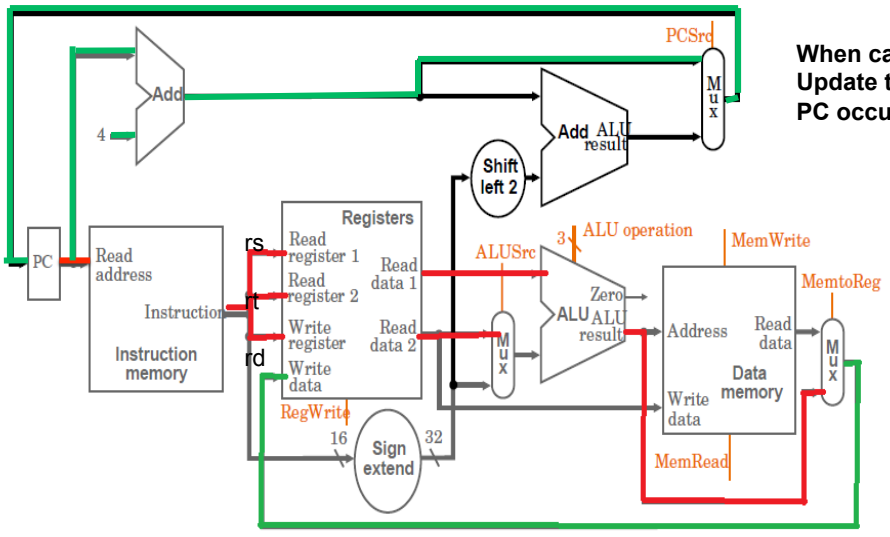


add $t1, $t2, $t3 **Next: Take Data of the two source registers**
***Send this through the ALU, Perform an operation as given in the instruction (add).**
***Produce a Result – output of the ALU**



add $t1, $t2, $t3

Next: This result needs to be Written to the Destination register.
***provide the result as Write Data to register file**
***Writing to the Register file occurs at end of the clock cycle.**

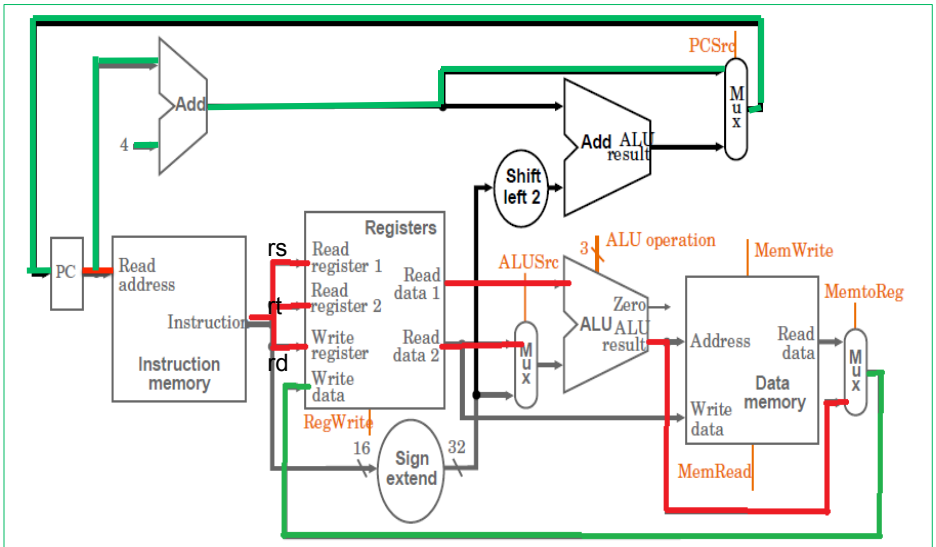


When can the Update to PC occur?

add $t1, $t2, $t3

PC :Can be updated right away in R-Format instructions. Computed value of PC+4 is ready.

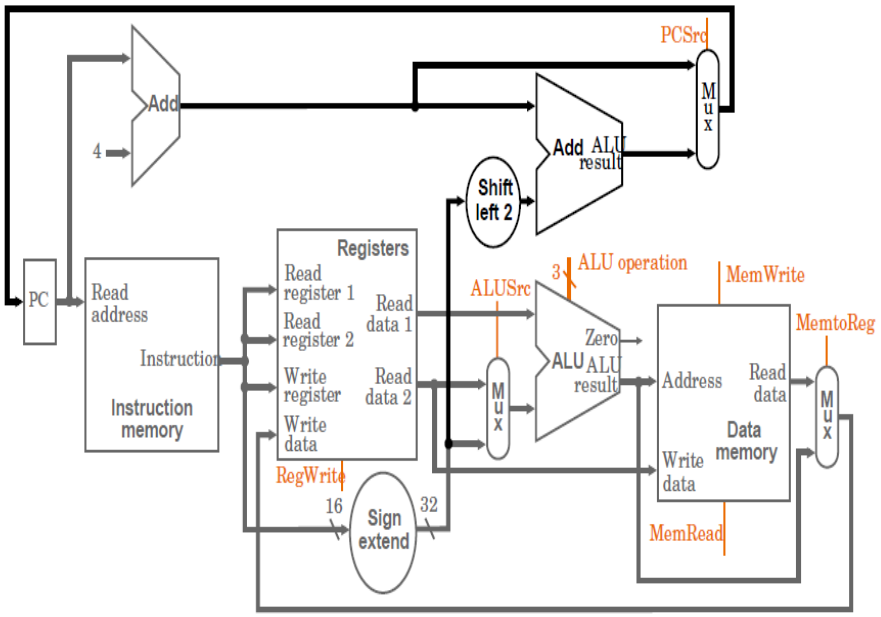
However, writes will occur at the end of the clock cycle. PC Register gets updated at end of clock cycle



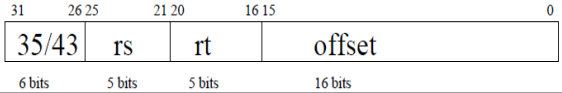
add $t1, $t2, $t3

Important: Control Line Bits (Select lines to Multiplexors, and other control lines) are essential in this process

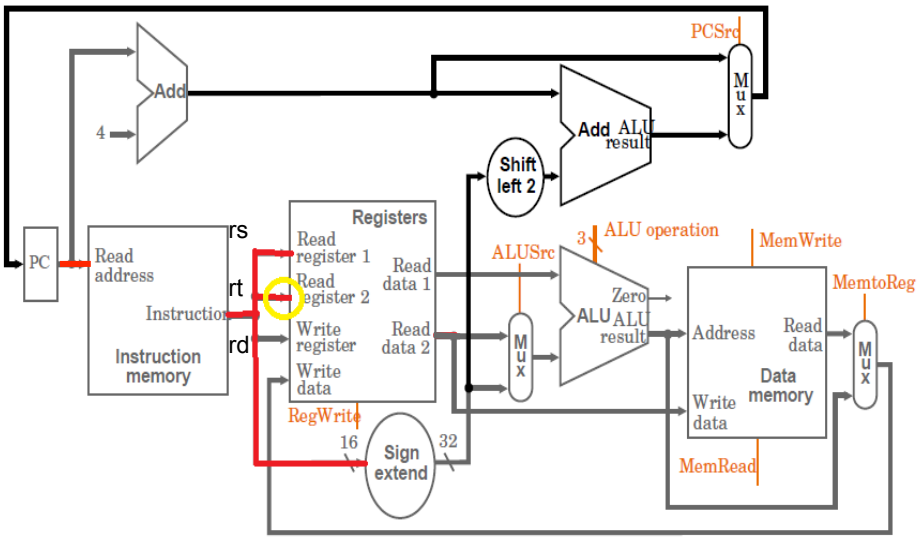
lw $t1, 100($t2)



Load/store: `lw $t1, 100($t2)` ⇒ `lw rt,100(rs)`



lw $t1, 100($t2)

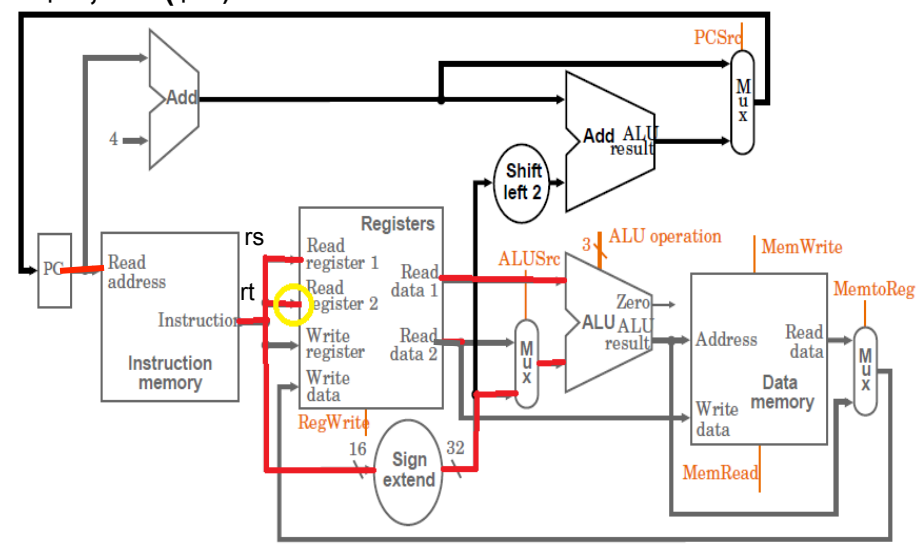


Now the **rt** register is the destination : Not a source register

Load/store: `lw $t1, 100($t2)` ⇒ `lw rt,100(rs)`

31–26	25–21	20–16	15–0
35/43	rs	rt	offset
6 bits	5 bits	5 bits	16 bits

lw $t1, 100($t2)

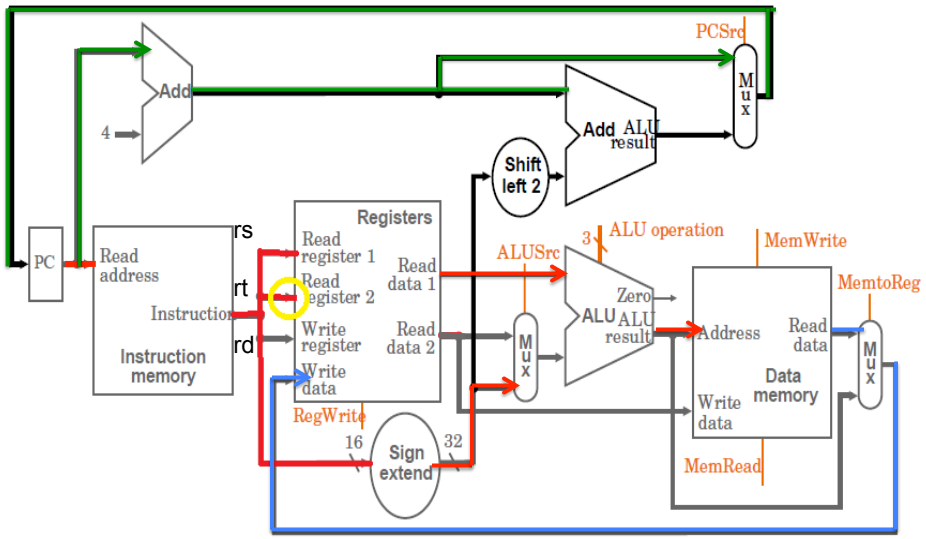


ALU: Will perform rs + sign extended offset
*Sign Extend lower 16 bits of instruction

Load/store: `lw $t1, 100($t2)` ⇒ `lw rt,100(rs)`

31–26	25–21	20–16	15–0
35/43	rs	rt	offset
6 bits	5 bits	5 bits	16 bits

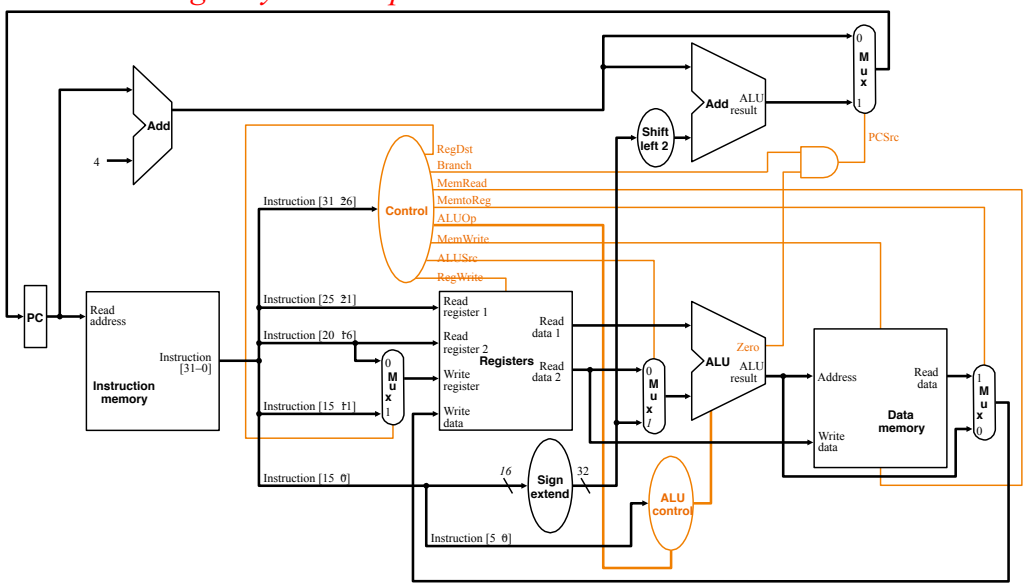
lw $t1, 100($t2)

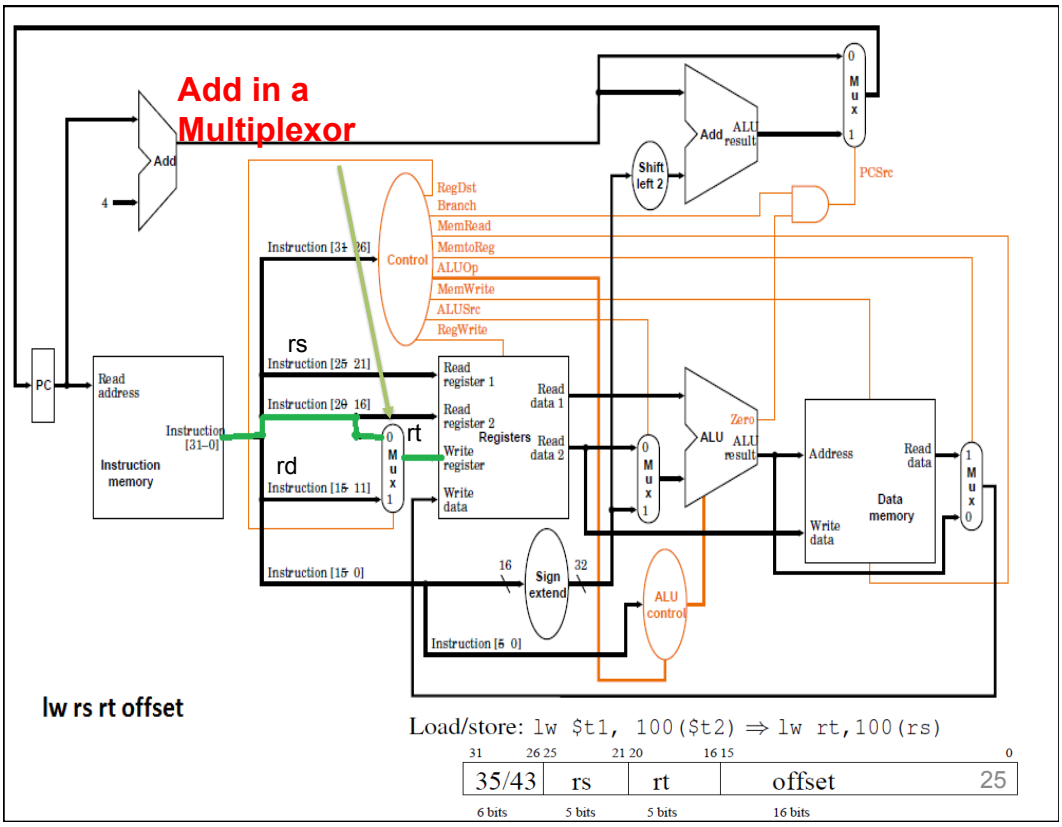


Data comes from Data Memory : At the address computed by The ALU.

Write Register: Wrong register being passed in.
Need more hardware to allow ***rt*** to also be a possible write register

Full Single Cycle Datapath + Control Unit





Meaning of Signals in Single-Cycle Datapath

Signal	Signal=0	Signal=1
RegDst	rt used	rd used
RegWrite	no effect	register written
ALUSrc	ALU B input from	immediate from
Branch*	no branch	instruction Branch
MemRead	no effect	memory read
MemWrite	no effect	memory written
MemToReg	reg write from ALU	reg write from memory

* Branch is ANDed with Zero from ALU to get PCsrc (Full version in Figure 4.16 of text.)

Opcode for every instruction is sent to Control Unit:

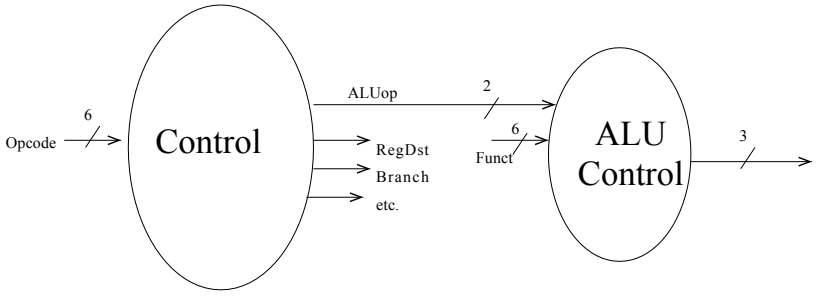
Type	Dec Opcode	Binary Opcode
R-format	0	000 000
lw	35	100 011
sw	43	101 011
beq	4	000 100

Control Unit will generate all the output bits as needed by the datapath for each individual instruction

Control Lines are set as needed by each instruction In order for the datapath to execute that instruction.

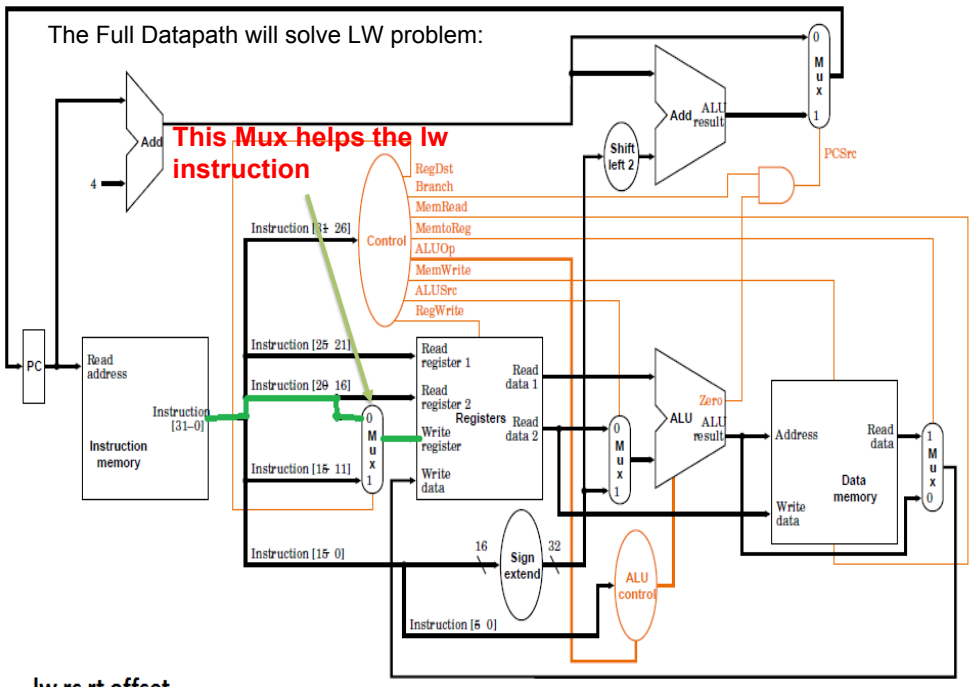
(we will come back to Control Unit Implementation soon)

Overview of Single-Cycle Control



- In our diagrams, the top 6 bits of the instruction (opcode) are always being sent into the Main Control Unit
- Multiple levels of control are conceptually simpler
- Smaller control units may also be faster
- Readings: Appendix D, section D.2

The Full Datapath will solve LW problem:

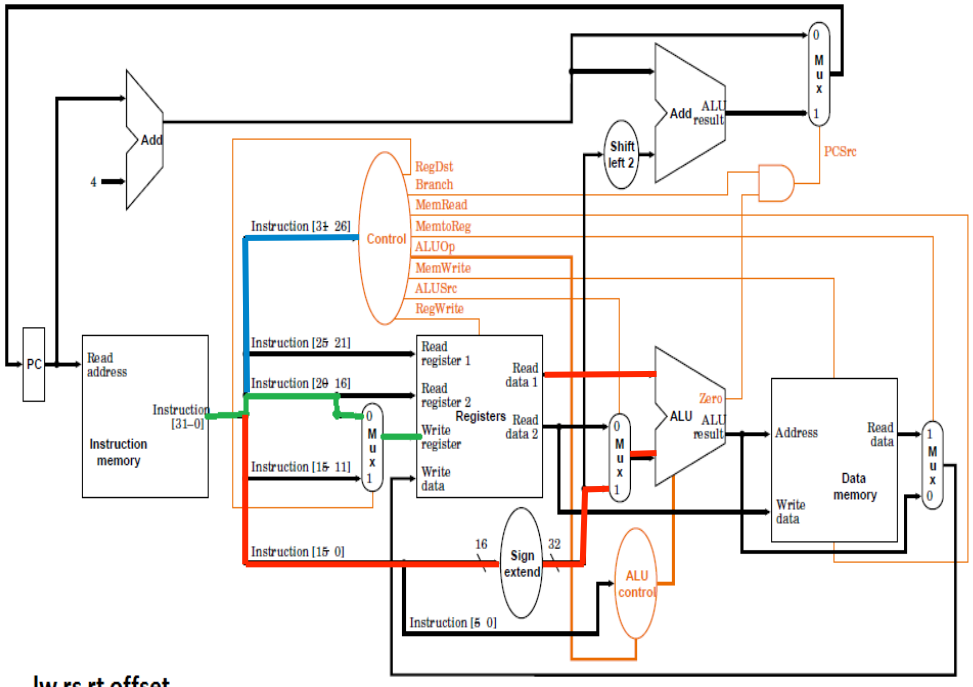


lw rs rt offset

Load/store: `lw $t1, 100($t2)` ⇒ `lw rt,100(rs)`

31–26	25–21	20–16	15–0
35/43	rs	rt	offset
6 bits	5 bits	5 bits	16 bits

29

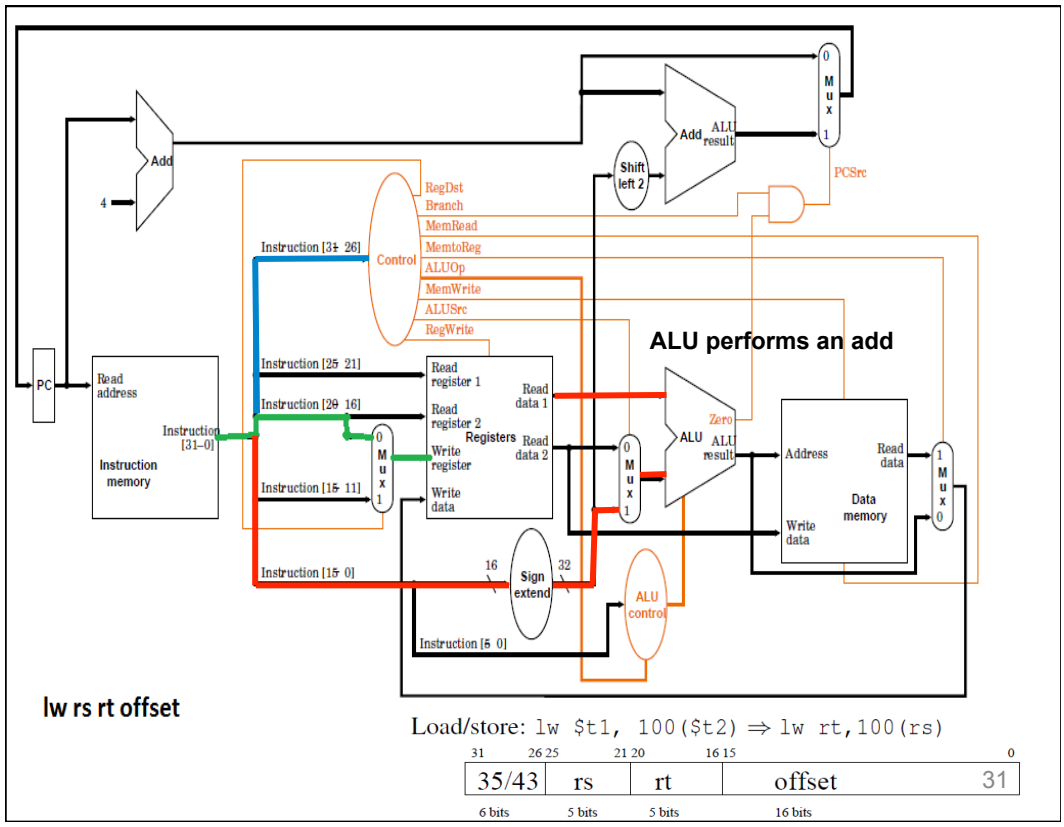


lw rs rt offset

Load/store: `lw $t1, 100($t2)` ⇒ `lw rt,100(rs)`

31–26	25–21	20–16	15–0
35/43	rs	rt	offset
6 bits	5 bits	5 bits	16 bits

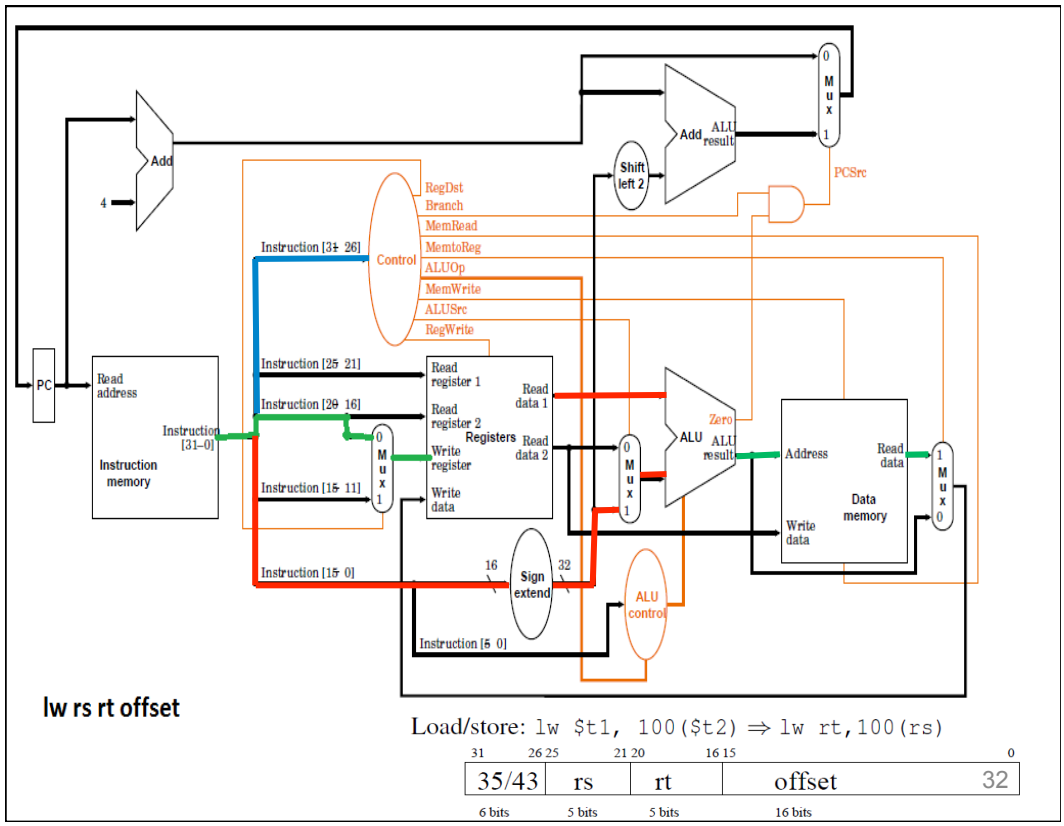
30



lw rs rt offset

Load/store: `lw $t1, 100($t2)` ⇒ `lw rt,100(rs)`

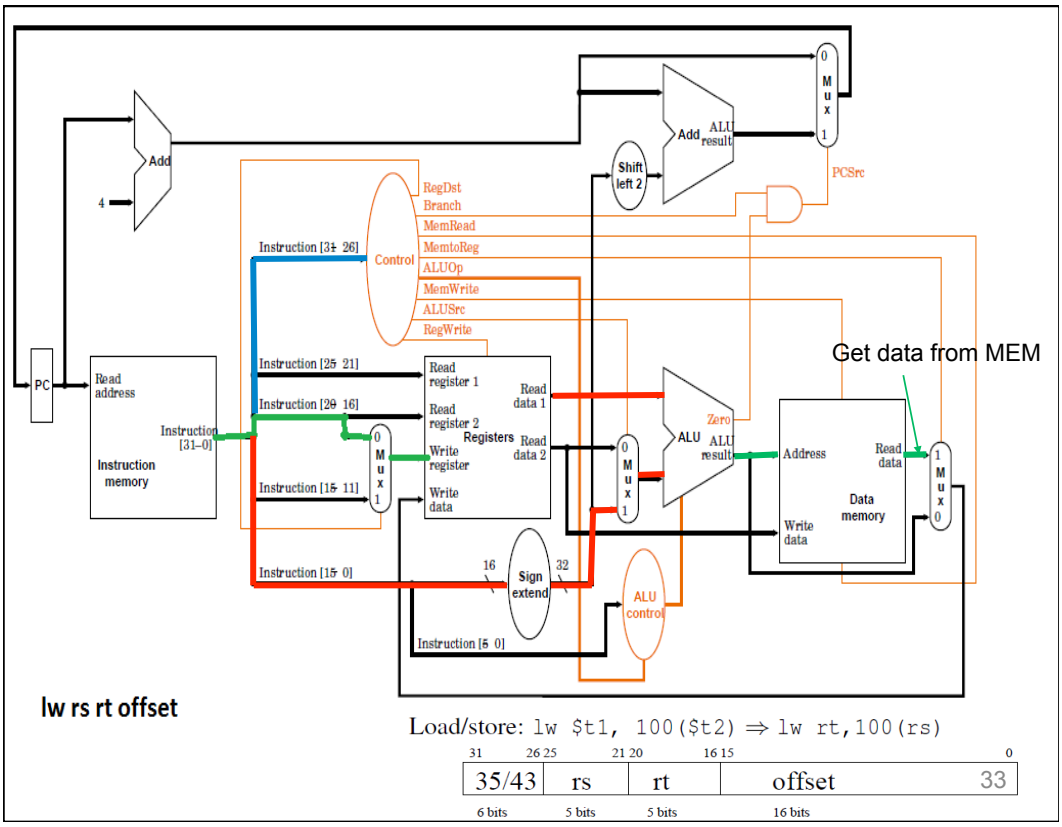
31–26	25–21	20–16	15–0
35/43	rs	rt	offset 31
6 bits	5 bits	5 bits	16 bits



lw rs rt offset

Load/store: `lw $t1, 100($t2)` ⇒ `lw rt,100(rs)`

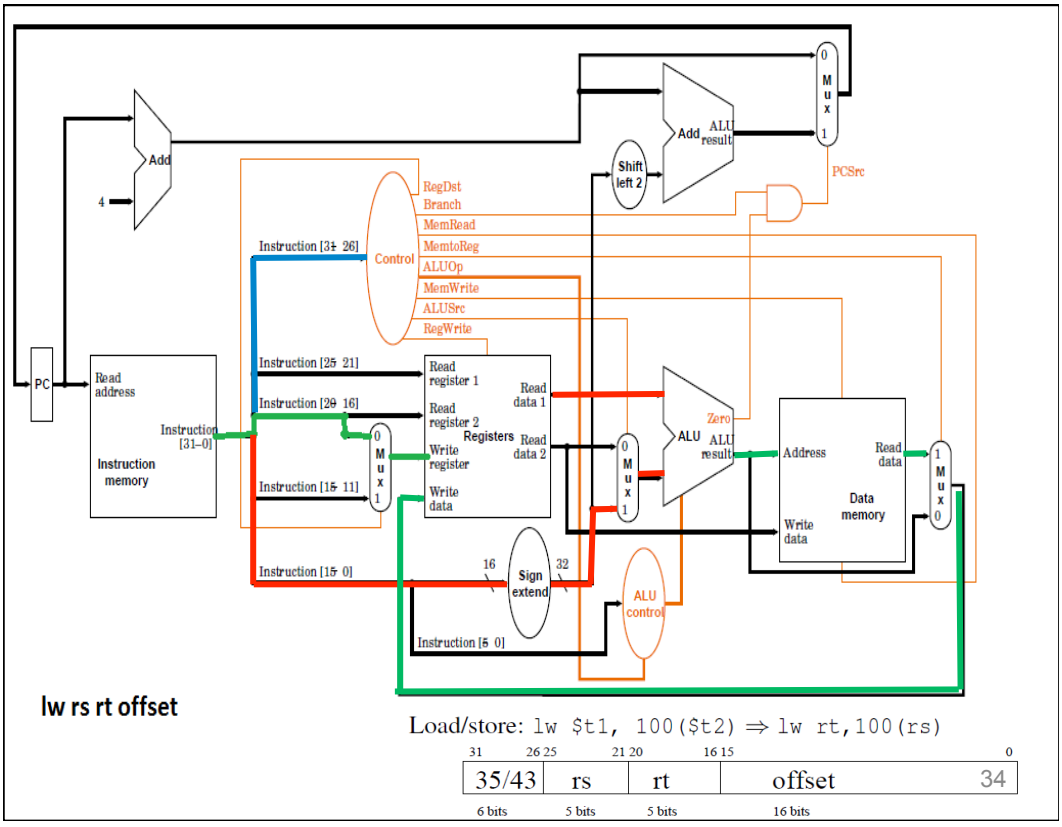
31–26	25–21	20–16	15–0
35/43	rs	rt	offset 32
6 bits	5 bits	5 bits	16 bits



lw rs rt offset

Load/store: `lw $t1, 100($t2)` ⇒ `lw rt,100(rs)`

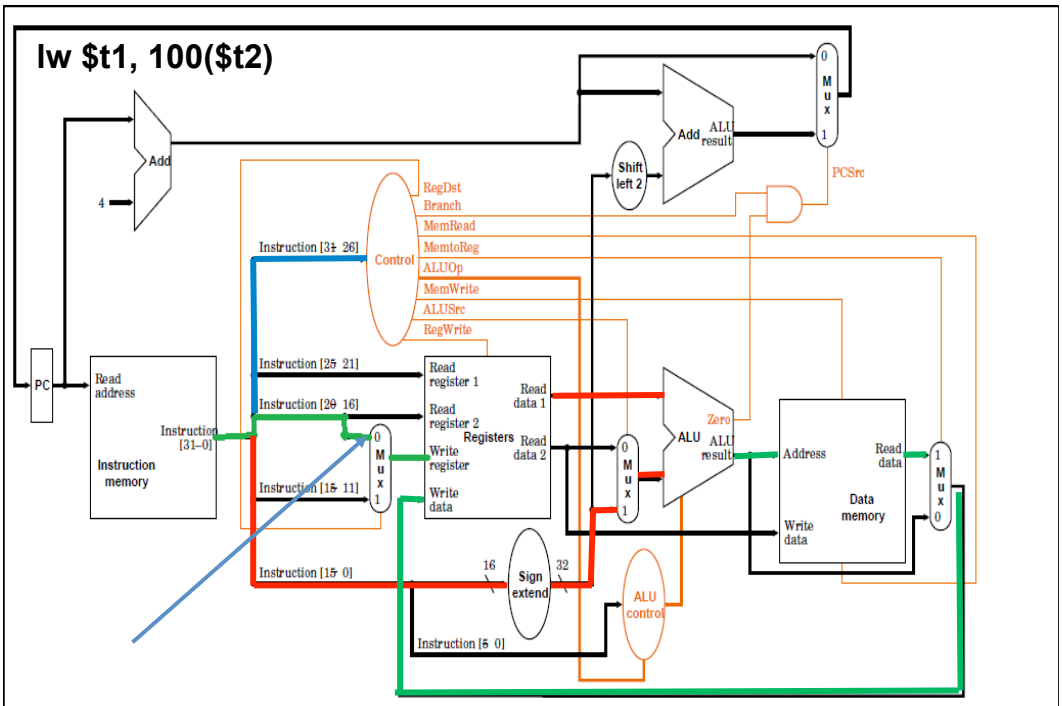
31–26	25–21	20–16	15–0
35/43	rs	rt	offset
6 bits	5 bits	5 bits	16 bits



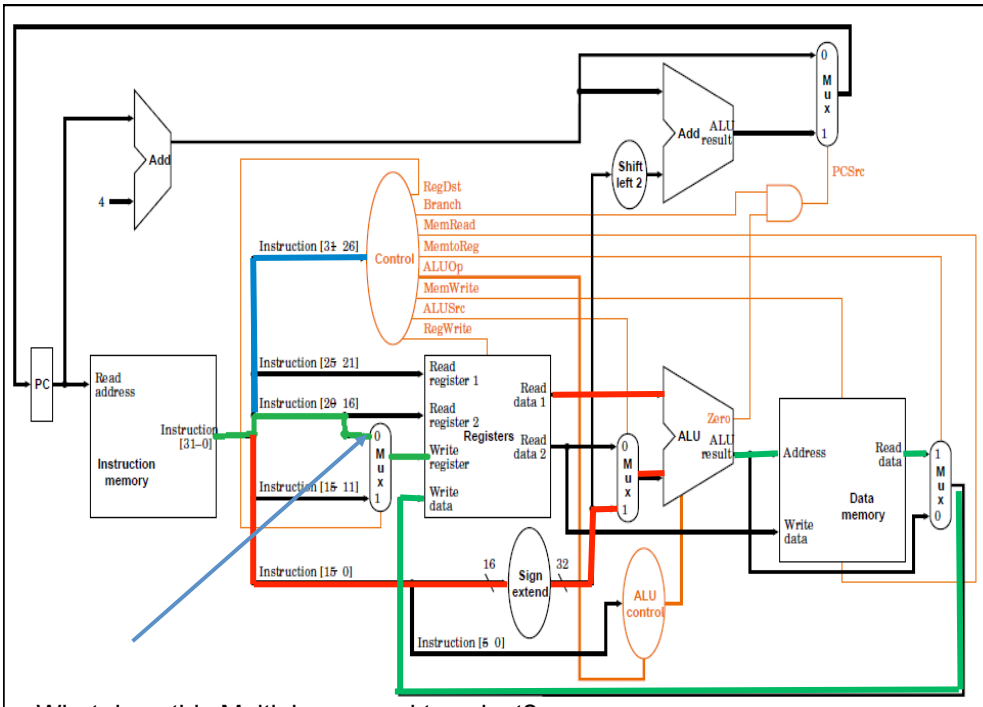
lw rs rt offset

Load/store: `lw $t1, 100($t2)` ⇒ `lw rt,100(rs)`

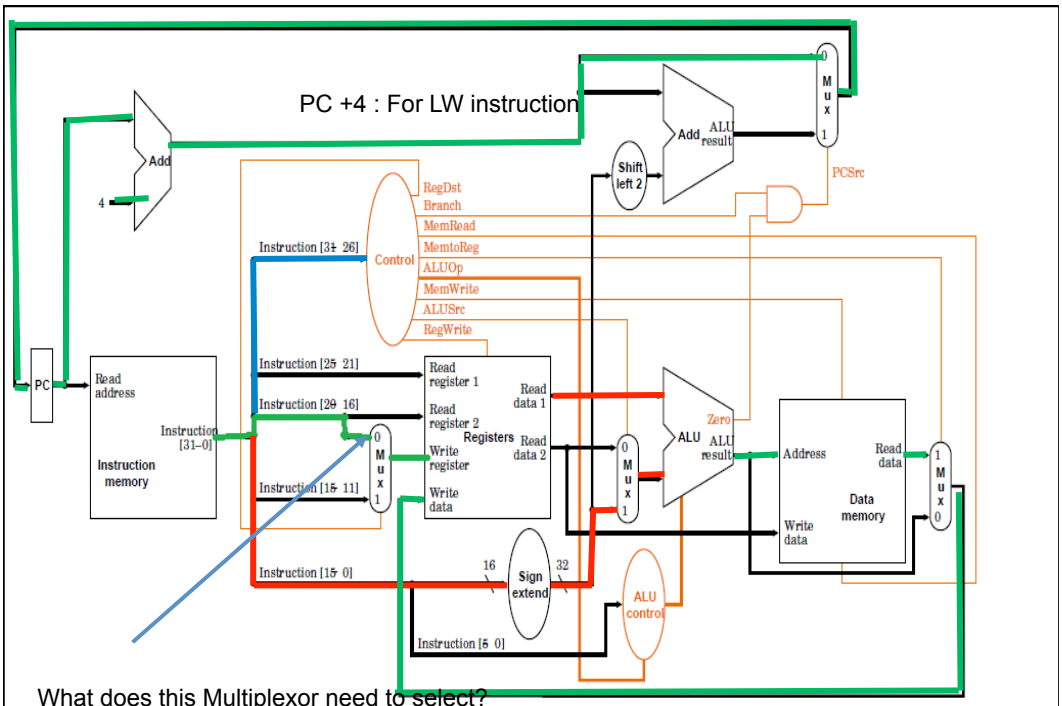
31–26	25–21	20–16	15–0
35/43	rs	rt	offset
6 bits	5 bits	5 bits	16 bits



What does this Multiplexor need to select?

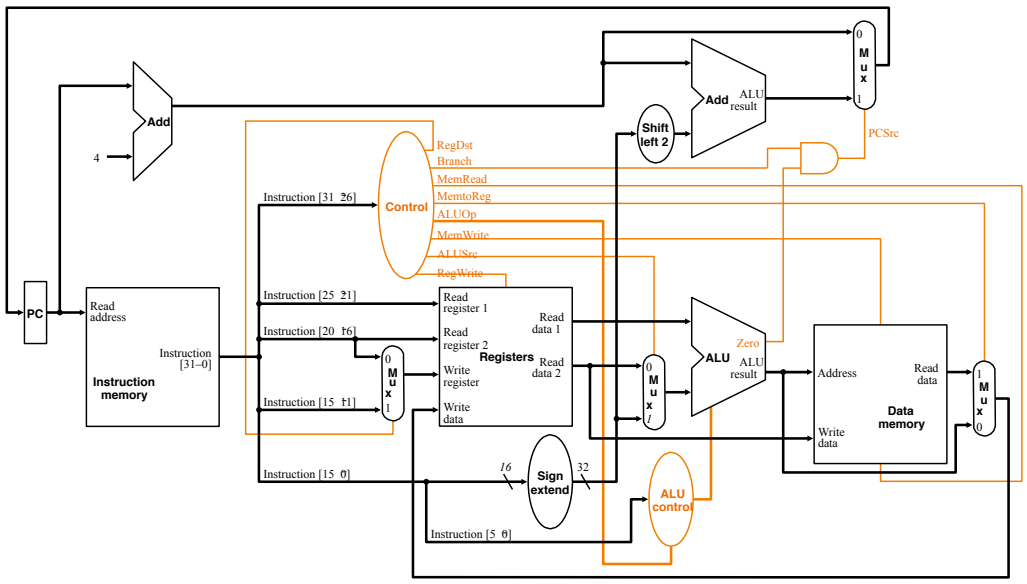


What does this Multiplexor need to select?
>>Zero
Option 1 is used for which instructions?

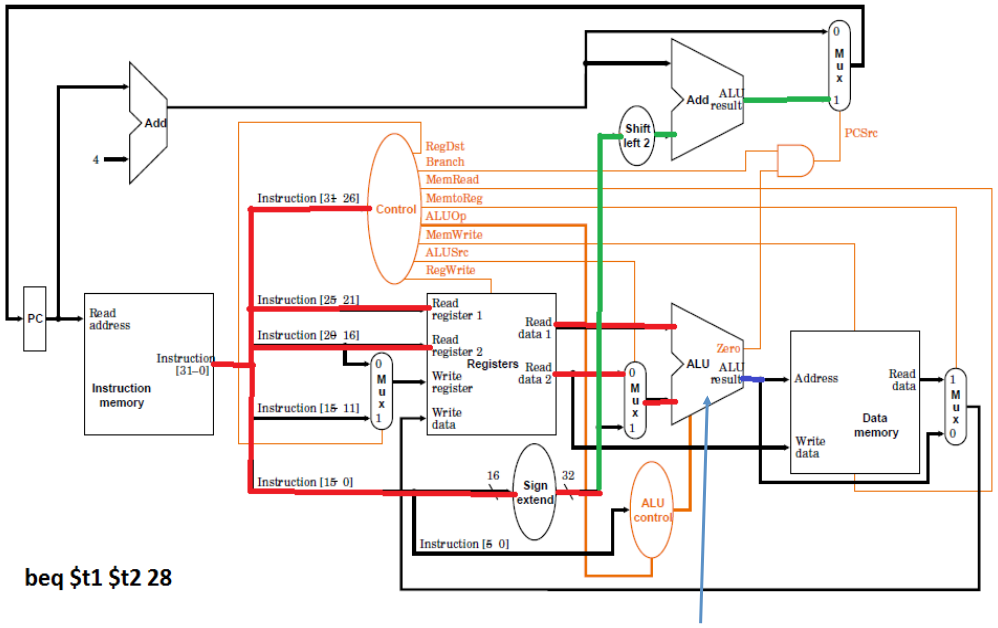


What does this Multiplexor need to select?
>>Zero
Option 1 is used for which instructions?
>>Rformat

Full Single Cycle Datapath



BRANCHING

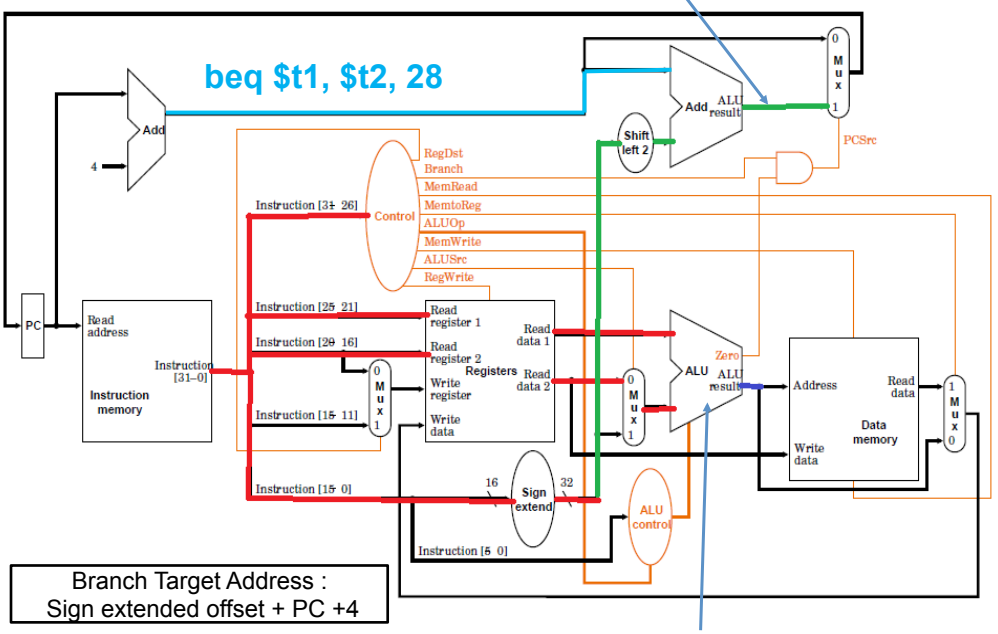


beq $t1 $t2 28

Branch instruction ALU will subtract
Two Source Registers

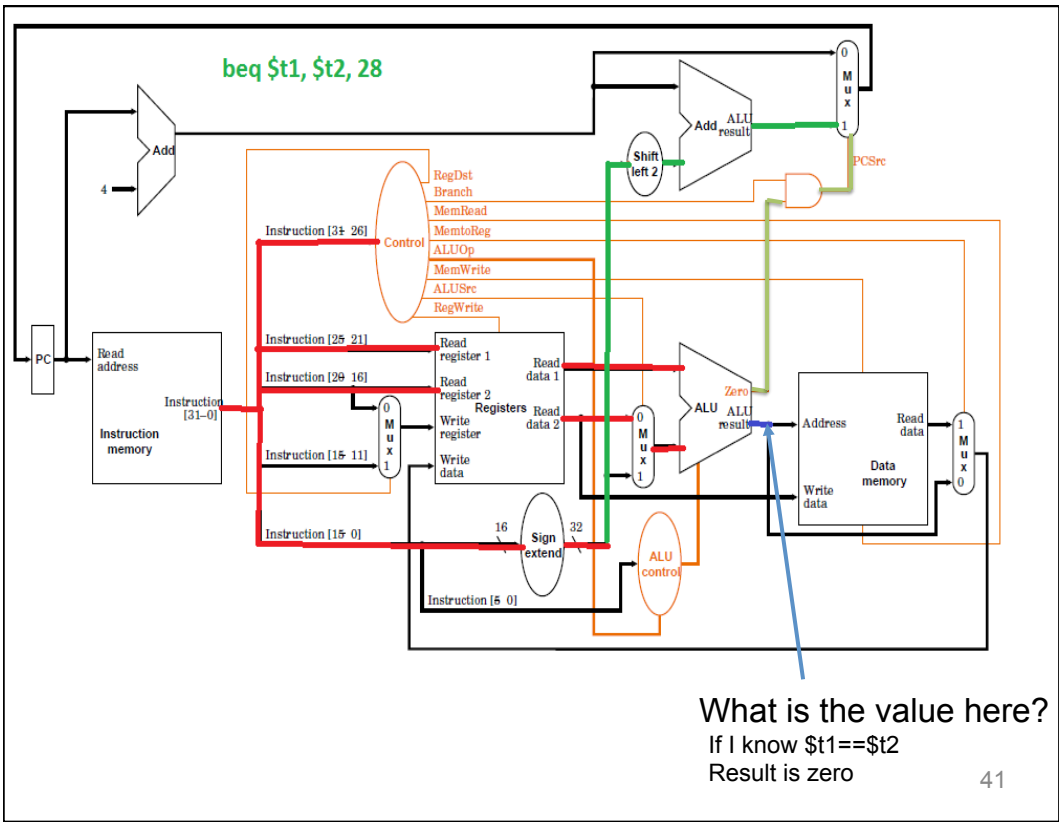
BRANCHING

Computation of Branch Target address



Branch Target Address :
Sign extended offset + PC +4

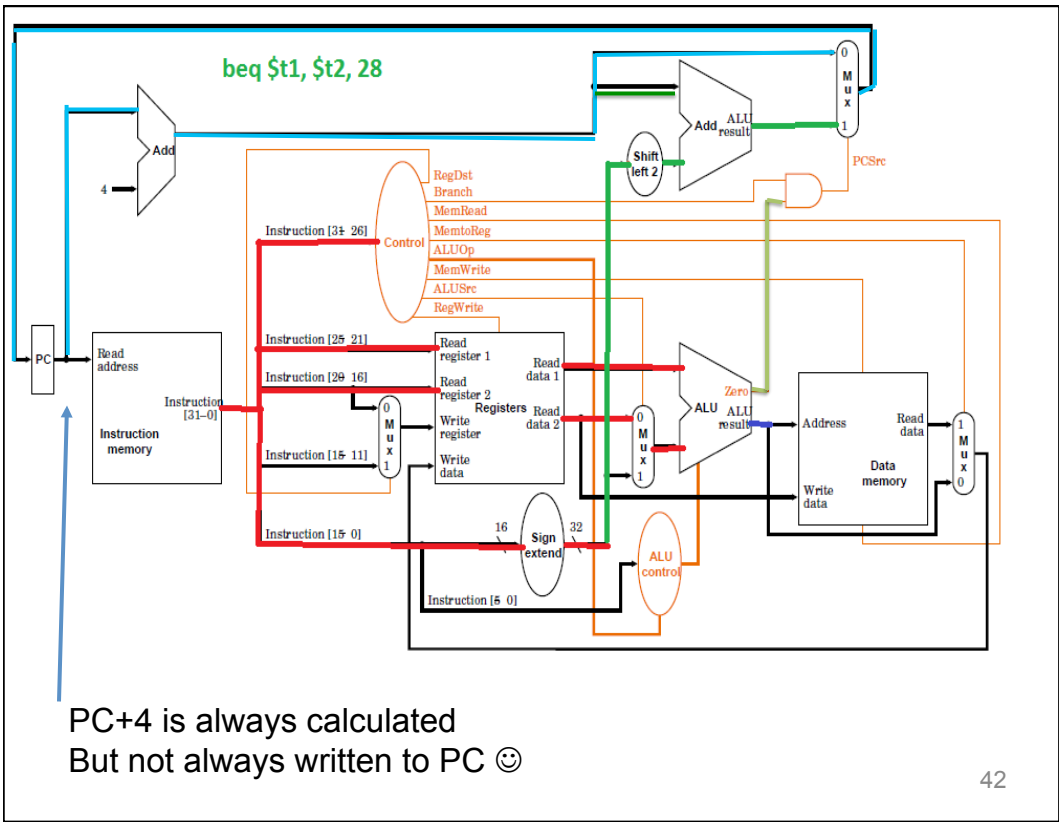
Branch instruction ALU will subtract
Two Source Registers



What is the value here?

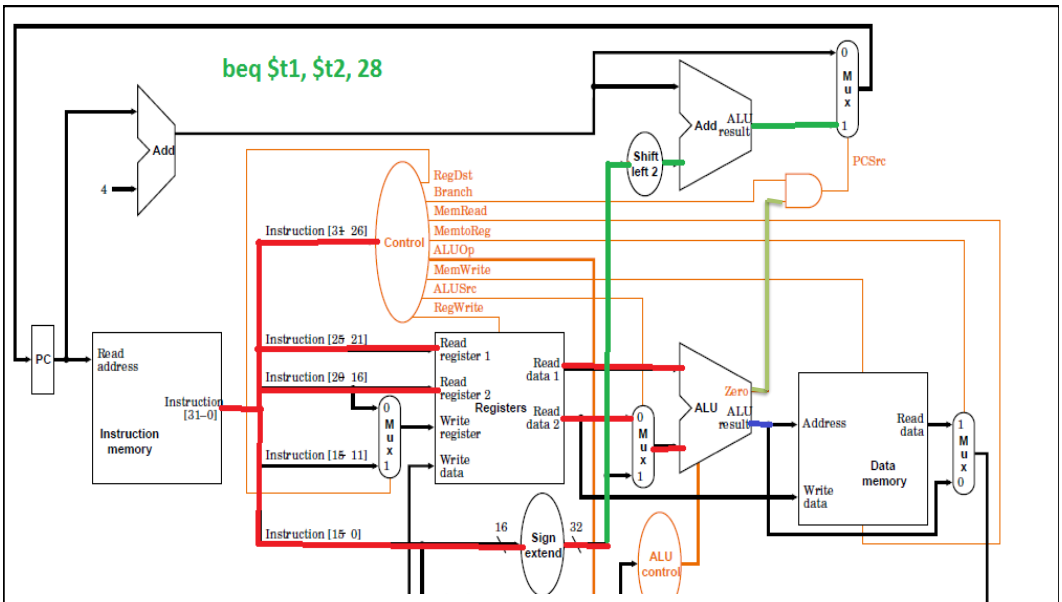
If I know $t1==$t2

Result is zero



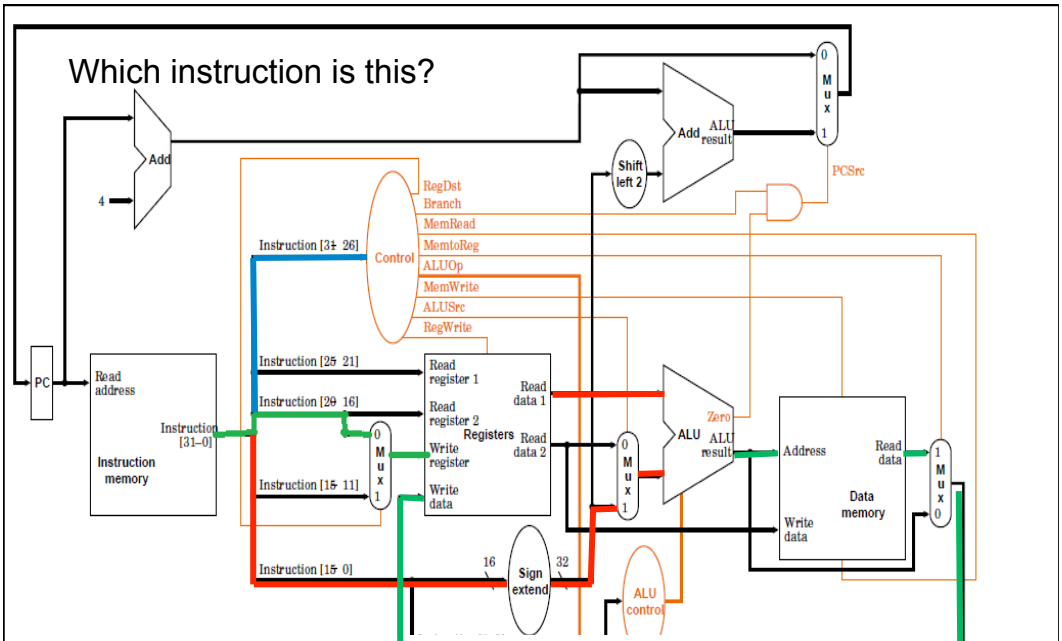
PC+4 is always calculated

But not always written to PC ☺



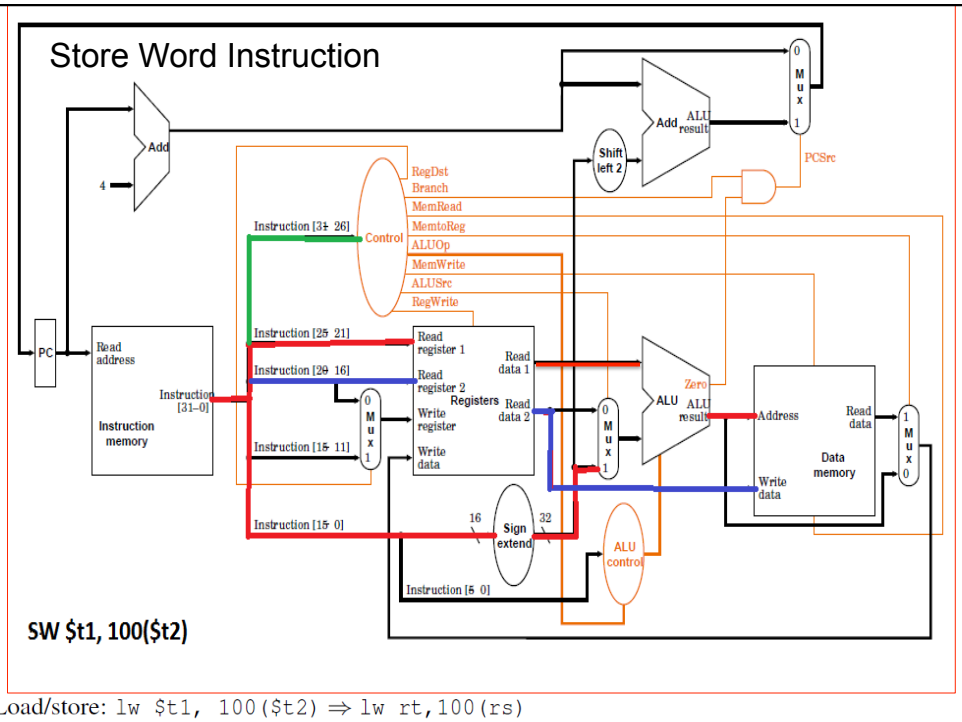
Type	Reg Dst	ALU Src	Mem ToReg	Reg Write	Mem Read	Mem Write	Branch	ALU op1	ALU op0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Tells you exactly what the control lines Need to be for each instruction



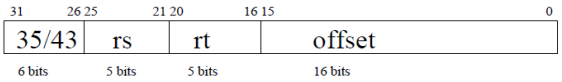
Type	Reg Dst	ALU Src	Mem ToReg	Reg Write	Mem Read	Mem Write	Branch	ALU op1	ALU op0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Store Word Instruction



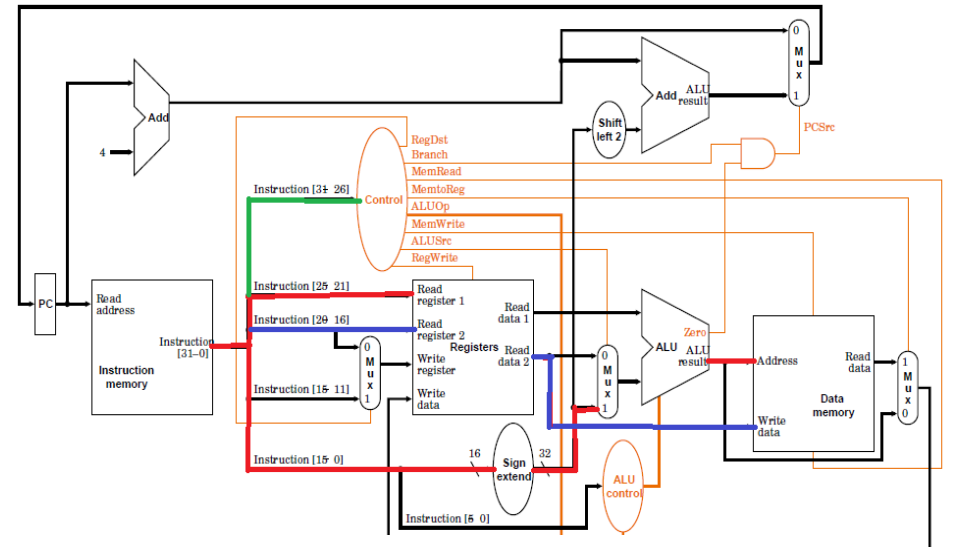
SW $t1, 100($t2)

Load/store: `lw $t1, 100($t2)` ⇒ `lw rt,100(rs)`

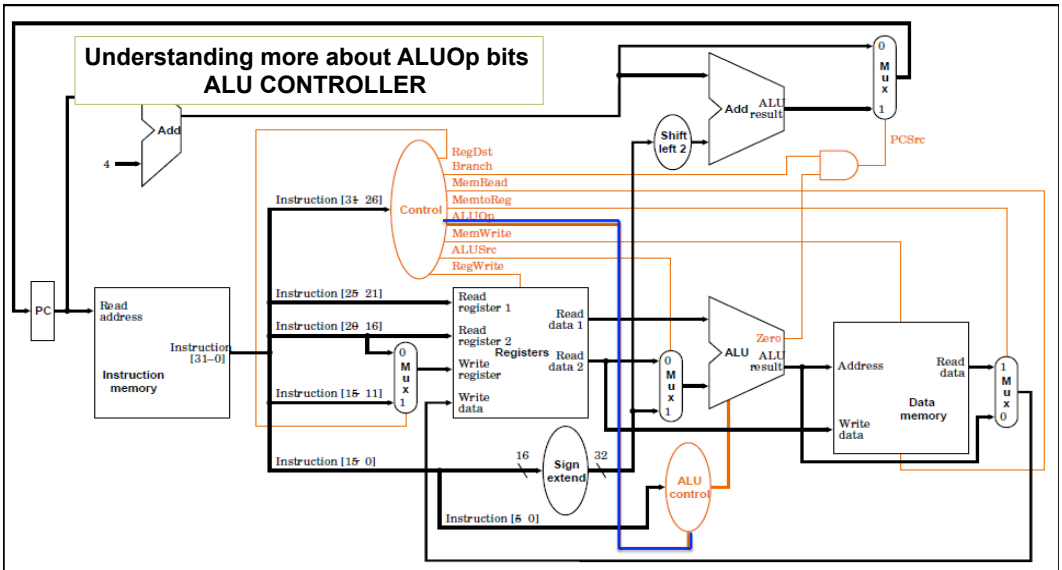


Note: PC = PC +4
Not always highlighted.

sw $t1, 100($t2)



Type	Reg Dst	ALU Src	Mem ToReg	Reg Write	Mem Read	Mem Write	Branch	ALU op1	ALU op0
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



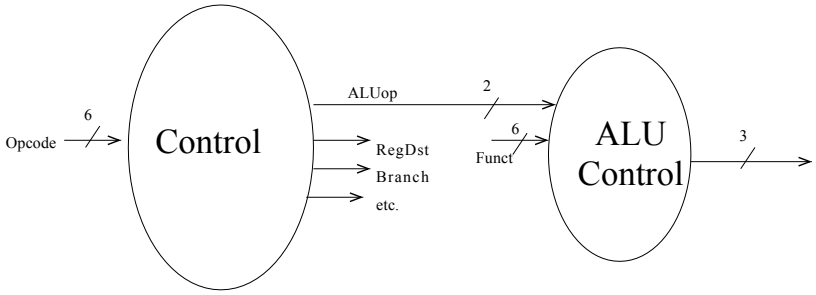
Type	Reg Dst	ALU Src	Mem ToReg	Reg Write	Mem Read	Mem Write	Branch	ALU op1	ALU op0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Main Control Unit
Sends a two bit message
To ALU Controller.

2bit message specifies
Some information about
The instruction

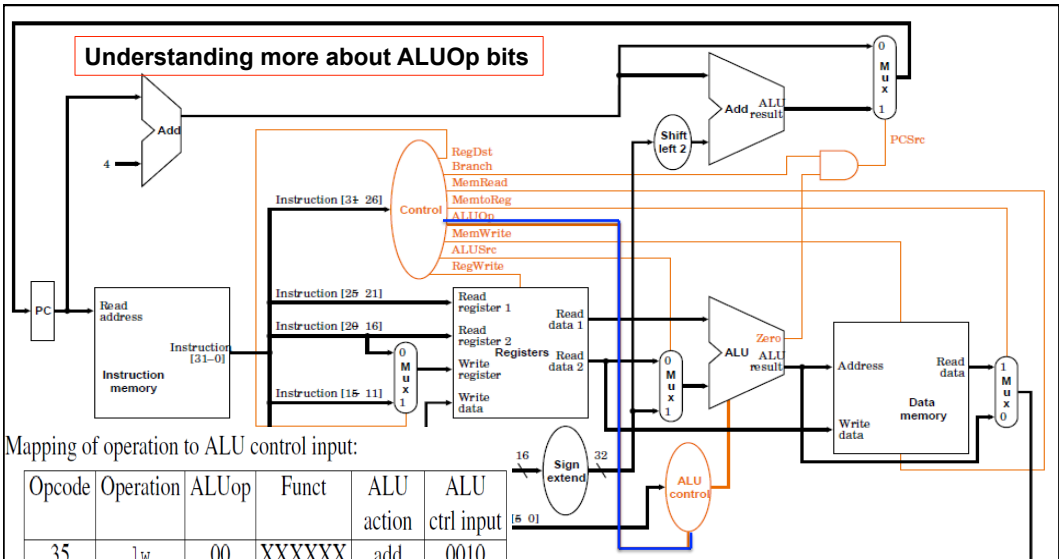
47

Purpose of ALU Control Unit



- Easier to have another Control Unit telling ALU
- What operation to perform.
- ALU Control will take a 2 bit message from Main Control
- These two bits will give information about what type of instruction it was.

Understanding more about ALUOp bits



Mapping of operation to ALU control input:

Opcode	Operation	ALUop	Funct	ALU action	ALU ctrl input
35	lw	00	XXXXXX	add	0010
43	sw	00	XXXXXX	add	0010
4	beq	01	XXXXXX	subtract	0110
0	add	10	100000	add	0010
0	sub	10	100010	subtract	0110
0	and	10	100100	AND	0000
0	or	10	100101	OR	0001
0	slt	10	101010	slt	0111

2 bit message says to do one of the following:

a) The instruction was an R-Format instruction, Therefore use the lower 6 function bits to decide what operation ALU should do.

b) The instruction was I-Format:

LW/SW : ALU should perform addition

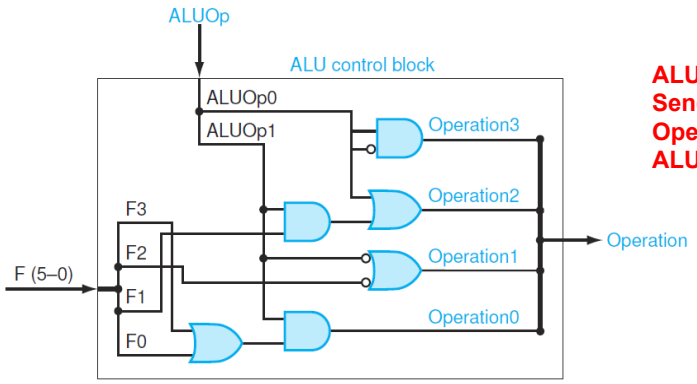
BEQ : ALU should perform subtraction

addi : ALU should perform addition…etc.

Mapping of operation to ALU control input:

Opcode	Operation	ALUop	Funct	ALU action	ALU ctrl input
35	lw	00	XXXXXX	add	0010
43	sw	00	XXXXXX	add	0010
4	beq	01	XXXXXX	subtract	0110
0	add	10	100000	add	0010
0	sub	10	100010	subtract	0110
0	and	10	100100	AND	0000
0	or	10	100101	OR	0001
0	slt	10	101010	slt	0111

ALUop		Funct field						Operation
ALUop1	ALUop0	F5	F4	F3	F2	F1	F0	3210
0	0	X	X	X	X	X	X	0010
X(0)	1	X	X	X	X	X	X	0110
1	X(0)	X	X	0	0	0	0	0010
1	X(0)	X	X	0	0	1	0	0110
1	X(0)	X	X	0	1	0	0	0000
1	X(0)	X	X	0	1	0	1	0001
1	X(0)	X	X	1	0	1	0	0111



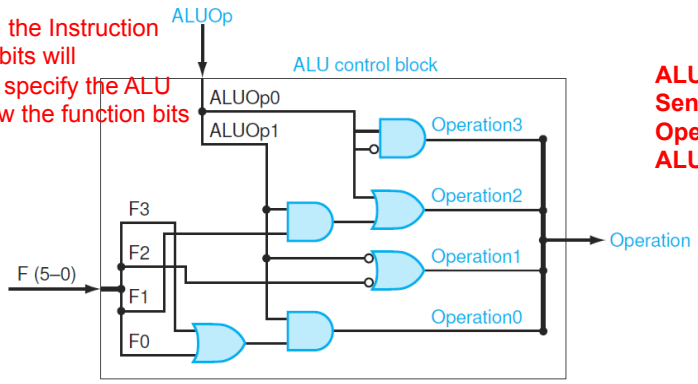
ALU control Sends 4 bit Operation code to ALU.

apping of operation to ALU control input:

Opcode	Operation	ALUop	Funct	ALU action	ALU ctrl input
35	lw	00	XXXXXX	add	0010
43	sw	00	XXXXXX	add	0010
4	beq	01	XXXXXX	subtract	0110
0	add	10	100000	add	0010
0	sub	10	100010	subtract	0110
0	and	10	100100	AND	0000
0	or	10	100101	OR	0001
0	slt	10	101010	slt	0111

ALUop		Funct field						Operation
ALUop1	ALUop0	F5	F4	F3	F2	F1	F0	3210
0	0	X	X	X	X	X	X	0010
X(0)	1	X	X	X	X	X	X	0110
1	X(0)	X	X	0	0	0	0	0010
1	X(0)	X	X	0	0	1	0	0110
1	X(0)	X	X	0	1	0	0	0000
1	X(0)	X	X	0	1	0	1	0001
1	X(0)	X	X	1	0	1	0	0111

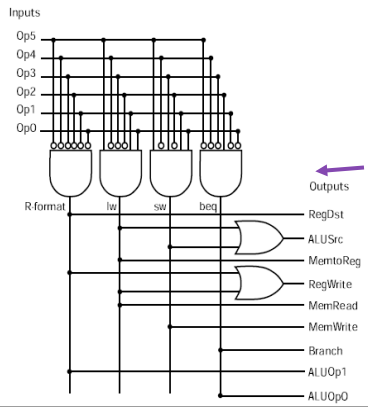
Depending on the Instruction
The ALUop 2 bits will
Either directly specify the ALU
Action, or allow the function bits
To do so



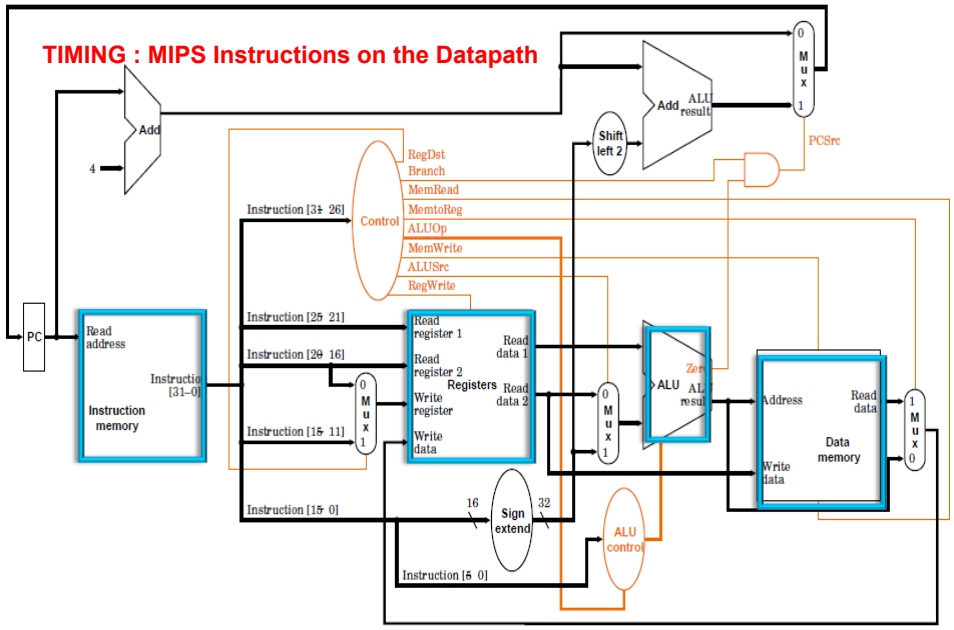
ALU control Sends 4 bit Operation code to ALU.

Type	Reg Dst	ALU Src	Mem ToReg	Reg Write	Mem Read	Mem Write	Branch	ALU op1	ALU op0
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

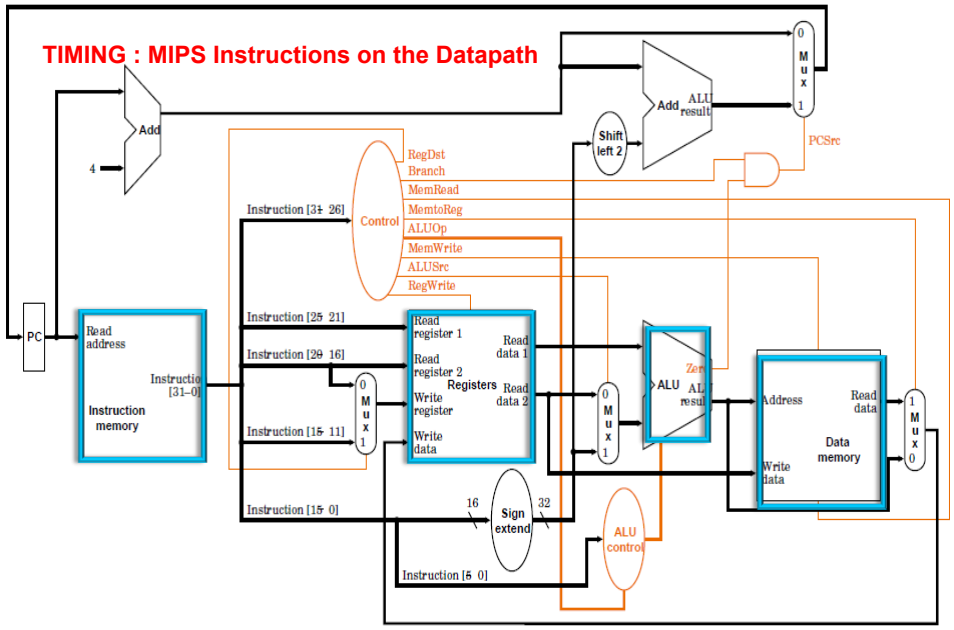
Type	Dec Opcode	Binary Opcode
R-format	0	000 000
lw	35	100 011
sw	43	101 011
beq	4	000 100



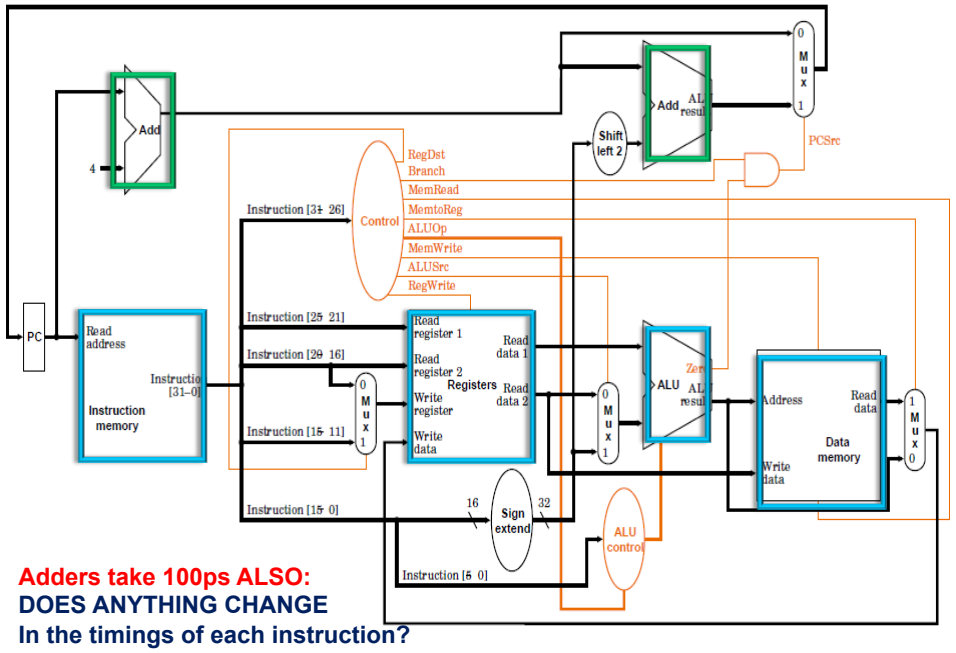
Depending on the Instruction opcode, Outputs will be set accordingly



- Suppose Memory units take 200ps (picoseconds) ALU take 100ps,
- Register File (w/r) 50ps
- Assume no time on other units (too minimal) (assume adders take 0 also)
- What is the execution time of each type of instruction?



- R-Format:
- Branch:
- LW:
- SW:



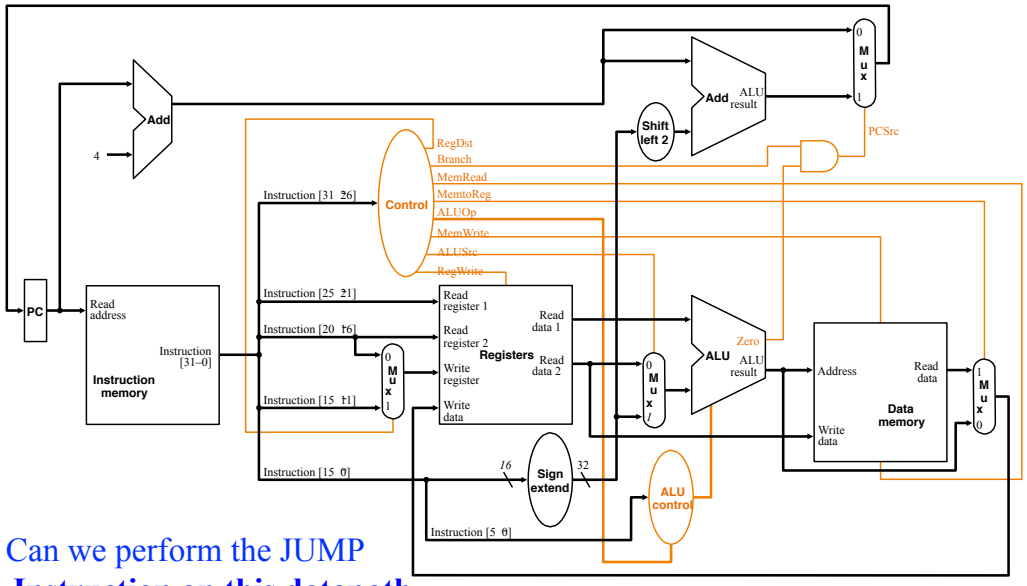
Adders take 100ps ALSO:
DOES ANYTHING CHANGE
In the timings of each instruction?

R-Format:
Branch:
LW:
SW:

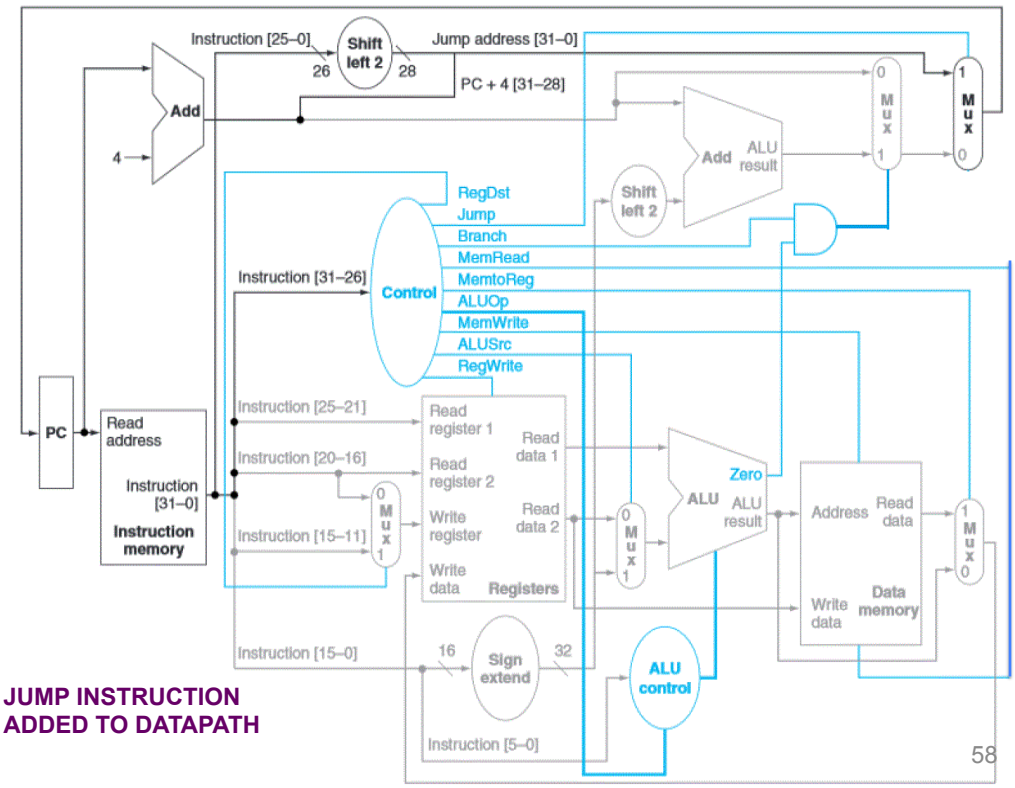
Performance of Single Cycle Machines

- Suppose memory units take 200 ps (picoseconds), ALUs 100 ps, register files 50 ps, no delay on other units
- Jumps take 200 ps, branches take 350 ps, R-format instructions 400 ps, stores 550 ps, loads 600 ps.
- Clock period must be increased to 600 ps or more
- Even worse when floating-point instructions are implemented
- Idea: Use this Basic Single Cycle Datapath
- Add Parallelism via Pipelining ☺

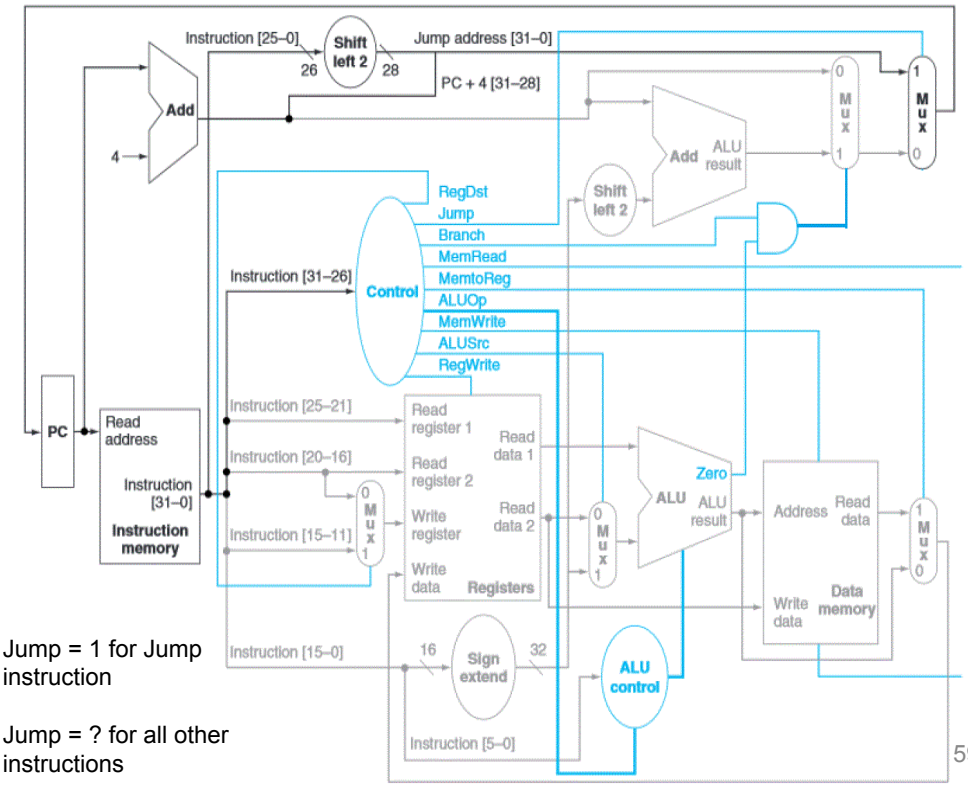
Full Single Cycle Datapath + Control Unit



Can we perform the JUMP
Instruction on this datapath
As it is ?



JUMP INSTRUCTION
ADDED TO DATAPATH



Jump = 1 for Jump instruction

Jump = ? for all other instructions

Adding a new instruction to the datapath:
Add Jump control line : An additional output from Control Unit

Type	Reg Dst	ALU Src	Mem ToReg	Reg Write	Mem Read	Mem Write	Branch	ALU op1	ALU op0	**Jump**
R-format	1	0	0	1	0	0	0	1	0	0
lw	0	1	1	1	1	0	0	0	0	0
sw	X	1	X	0	0	1	0	0	0	0
beq	X	0	X	0	0	0	1	0	1	0
Jump	**X**	**X**	**X**	**0**	**0**	**0**	**__**	**X**	**X**	**1**

What is value of Branch Bit for the JUMP Instruction

Type	Reg Dst	ALU Src	Mem ToReg	Reg Write	Mem Read	Mem Write	Branch	ALU op1	ALU op0	Jump
R-format	1	0	0	1	0	0	0	1	0	0
lw	0	1	1	1	1	0	0	0	0	0
sw	X	1	X	0	0	1	0	0	0	0
beq	X	0	X	0	0	0	1	0	1	0
Jump	**X**	**X**	**X**	**0**	**0**	**0**	**__**	**X**	**X**	**1**

What is value of Branch Bit Control Line for the JUMP Instruction

A) 1 B) 0 C) X

Type	Reg Dst	ALU Src	Mem ToReg	Reg Write	Mem Read	Mem Write	Branch	ALU op1	ALU op0	Jump
R-format	1	0	0	1	0	0	0	1	0	0
lw	0	1	1	1	1	0	0	0	0	0
sw	X	1	X	0	0	1	0	0	0	0
beq	X	0	X	0	0	0	1	0	1	0
Jump	**X**	**X**	**X**	**0**	**0**	**0**	**__**	**X**	**X**	**1**

Why Concatenate top 4 bits of PC to Jump Target Address 28 bits?

Jump Target address is PC relative, in the sense that
We have to stay within the Range of Program memory that we are in.
We cannot jump around anywhere in Program memory.

We have 26bit range x4 (word aligned): 2^{28} addresses we can jump to Pretty big ☺

Multiply by 4 is handled in the hardware

How long Does Jump Instruction take?

Case 1:
Jump: Read from Instruction memory **(200ps)**
Shift Left x2 : Minimal Time **(0)**
Write to PC: Minimal Time **(0)**
Control Unit: Minimal Time **(0)**

Total Time for Jump: 200ps

Case 2: ?
Jump: Read from Instruction memory **(200ps)**
Shift Left x2 : Minimal Time **(10)**
Write to PC: Minimal Time **(0)**
Control Unit: Minimal Time **(10)**

A)200 B) 220 C) 210 D)NONE

Modifying the datapath

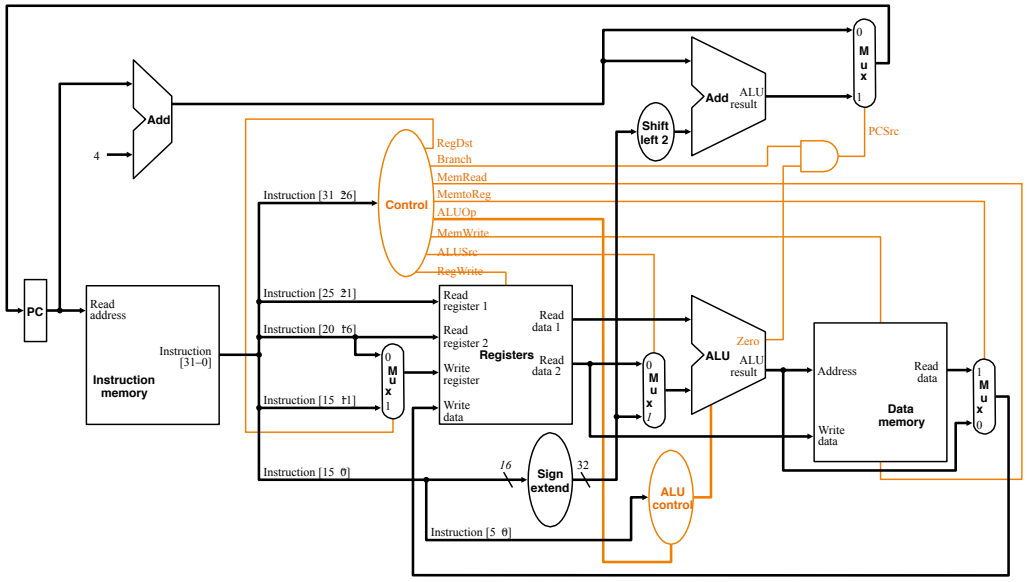
- Normally design complete datapath for all instructions together.
- Various ways to modify datapath. The following is one approach for adding a new assembly instruction:
  1. Determine what datapath is needed for new command
  2. Check if any components in current datapath can be used
  3. Wire in components of new datapath into existing datapath Probably requires MUXes
  4. Add new control signals to Control units
  5. Adjust old control signals to account for new command

Modifying the datapath

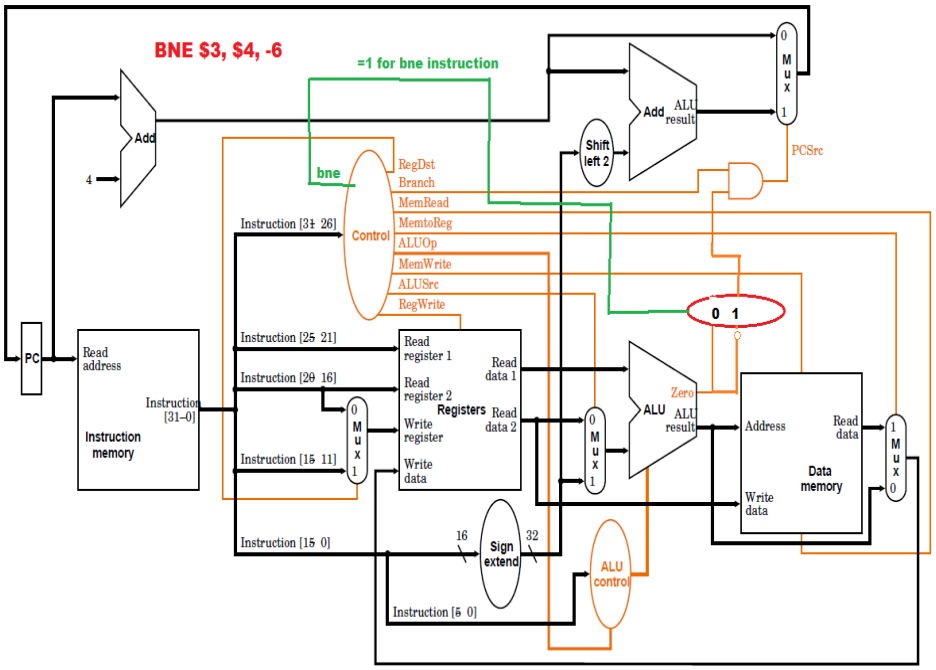
- Subsequent Examples:

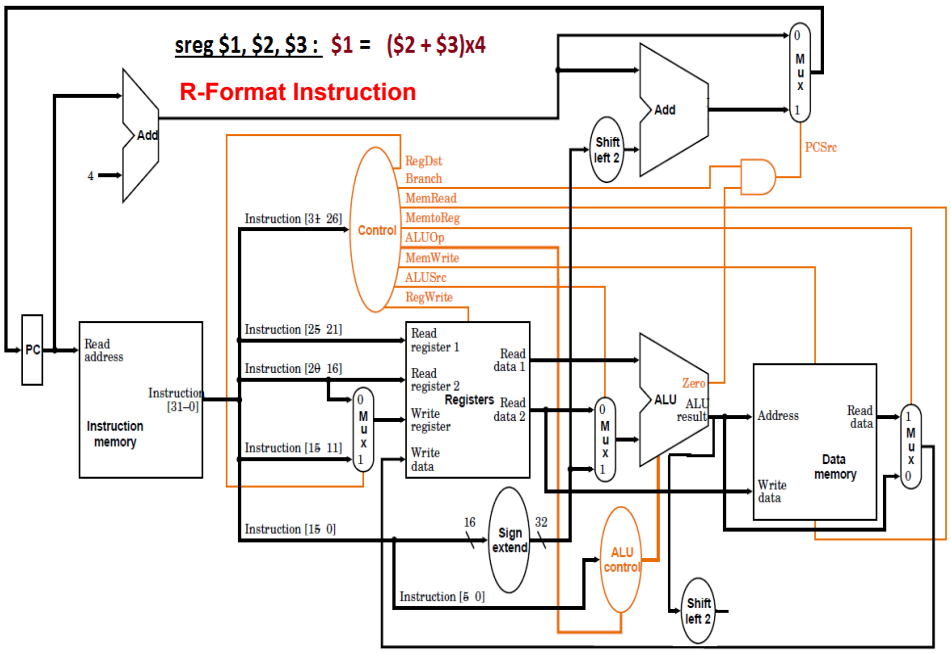
  1. Bne: Branch not equal. Does the datapath correctly perform this instruction. If not what needs to be changed
  2. Sreg: Set register →sets $rd to the ($rs+$rt)x4
  3. Jrel : Jump relative → jump to new address provided by ($rs x 4) + PC + 4
  4. Note: Always reuse as much of datapath as possible, if it is not inefficient to do so.

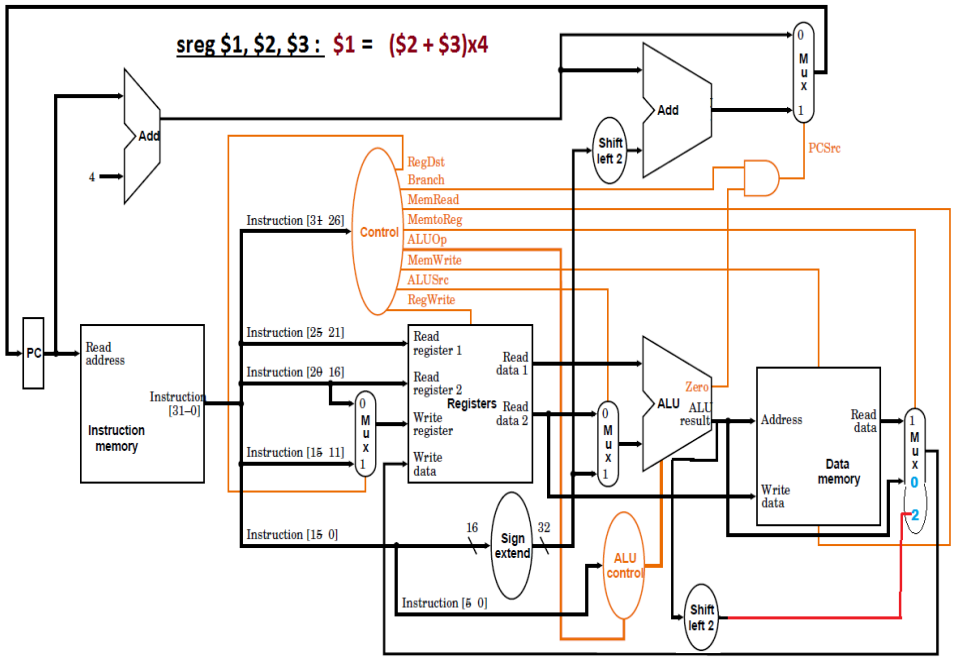
Examine Existing Datapath:

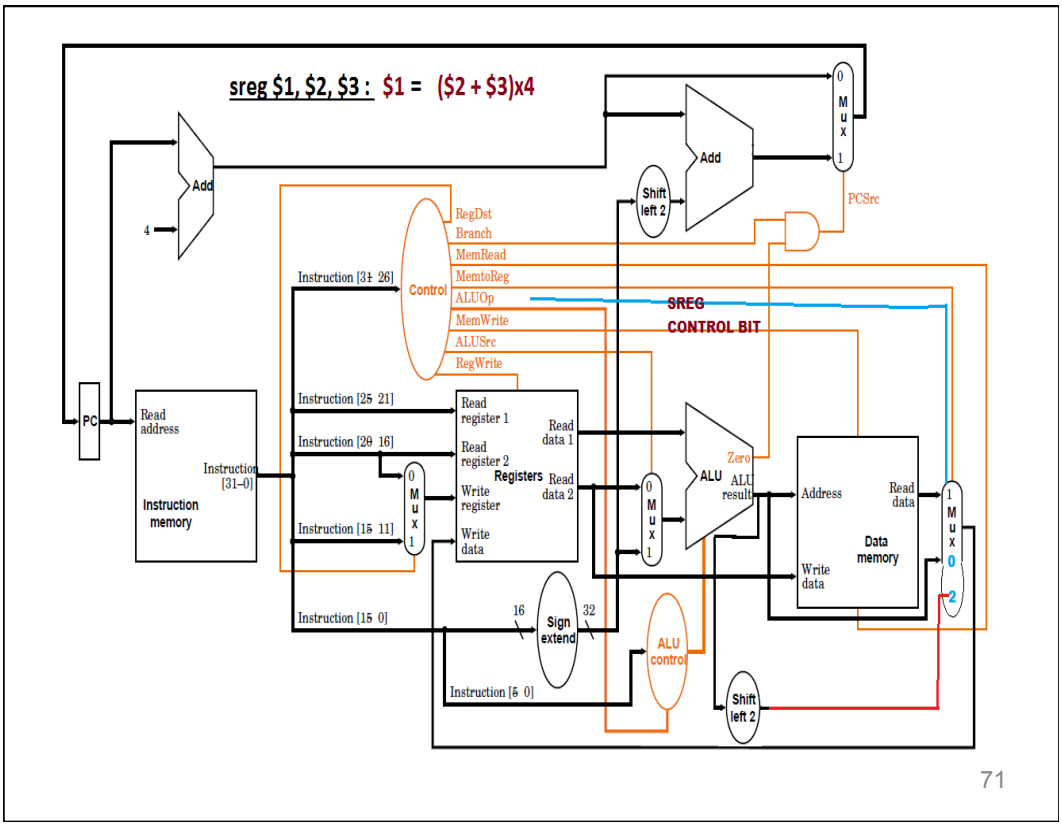


Can the new instructions be implemented
Without any changes.

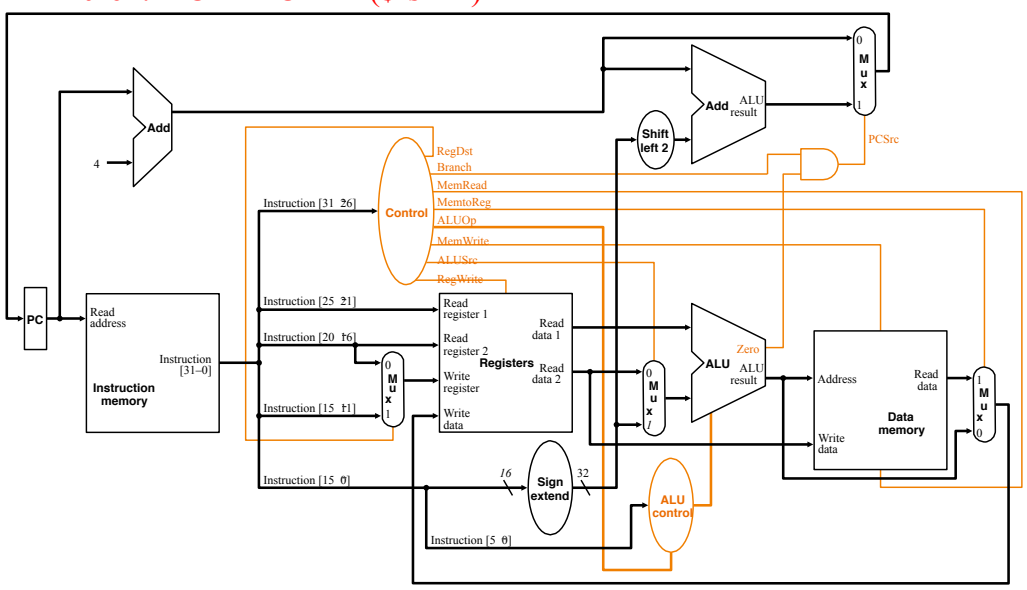



sreg $1, $2, $3 : $1 = ($2 + $3)x4
R-Format Instruction
Add
4
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [31–26]
Control
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
M u x
0
1
Shift left 2
Add
PCSrc
Zero
ALU
ALU result
Address
Read data
Data memory
Write data
16
Sign extend
32
ALU control

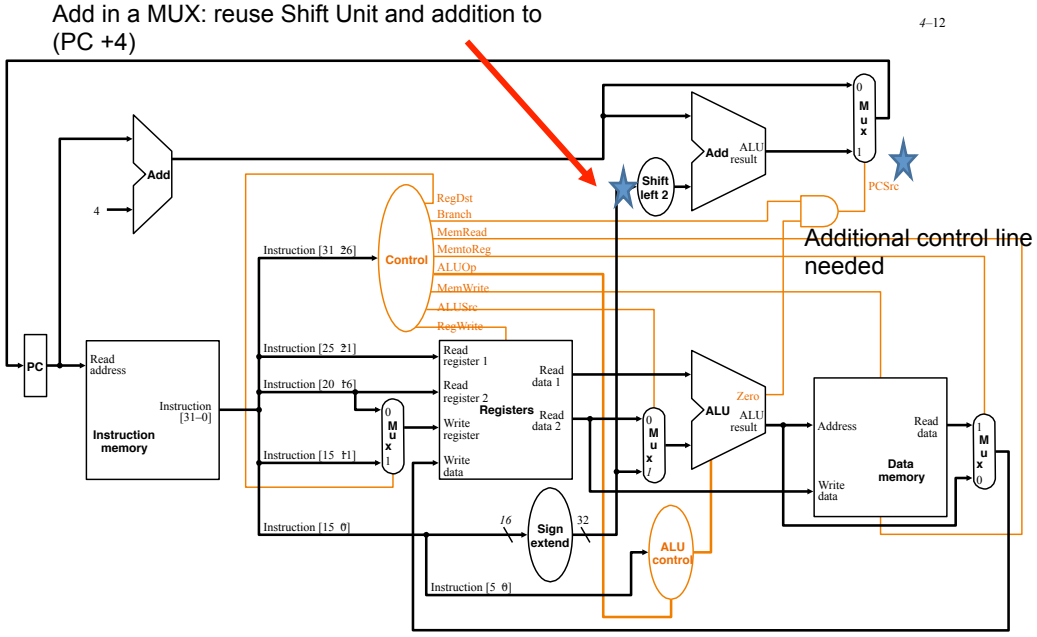

sreg $1, $2, $3 : $1 = ($2 + $3)x4
Add
4
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [31–26]
Control
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
M u x
0
1
Shift left 2
Add
PCSrc
Zero
ALU
ALU result
Address
Read data
Data memory
Write data
16
Sign extend
32
ALU control
2



Jrel : PC ← PC+4 + ($rs x 4)



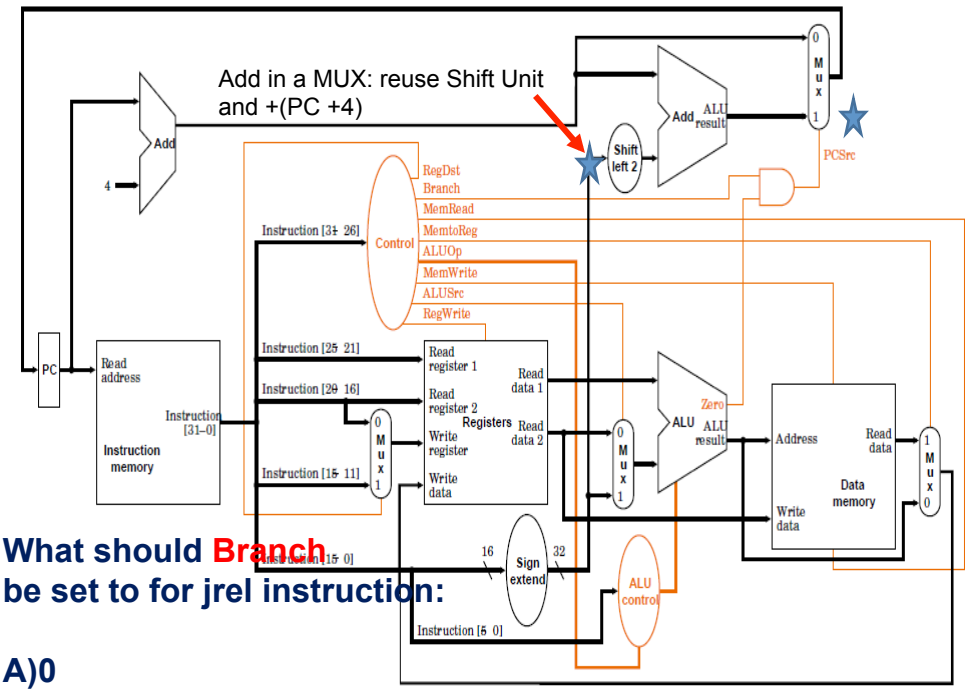
Add in a MUX: reuse Shift Unit and addition to (PC +4)



Jrel : PC ← PC+4 + ($rs x 4) :
control line call it **jrel**

When we modify the datapath and add a new instruction,
We also *usually* need to add a Control Line coming out of the Control Unit

Type	Reg Dst	ALU Src	Mem ToReg	Reg Write	Mem Read	Mem Write	Branch	ALU op1	ALU op0	**Jrel**
R-format	1	0	0	1	0	0	0	1	0	**0**
lw	0	1	1	1	1	0	0	0	0	**0**
sw	X	1	X	0	0	1	0	0	0	**0**
beq	X	0	X	0	0	0	1	0	1	**0**
Jrel	**X**	**X**	**X**	**0**	**0**	**0**	**----**	**X**	**X**	**1**



What should Branch be set to for jrel instruction:

A)0
B)1
C)X

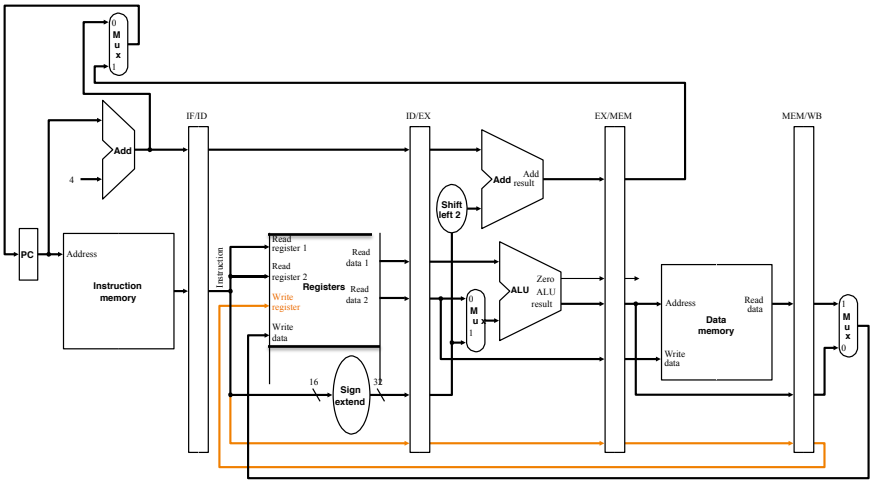
Multicycle Computer

Datapaths

Multicycle Processor Implementations

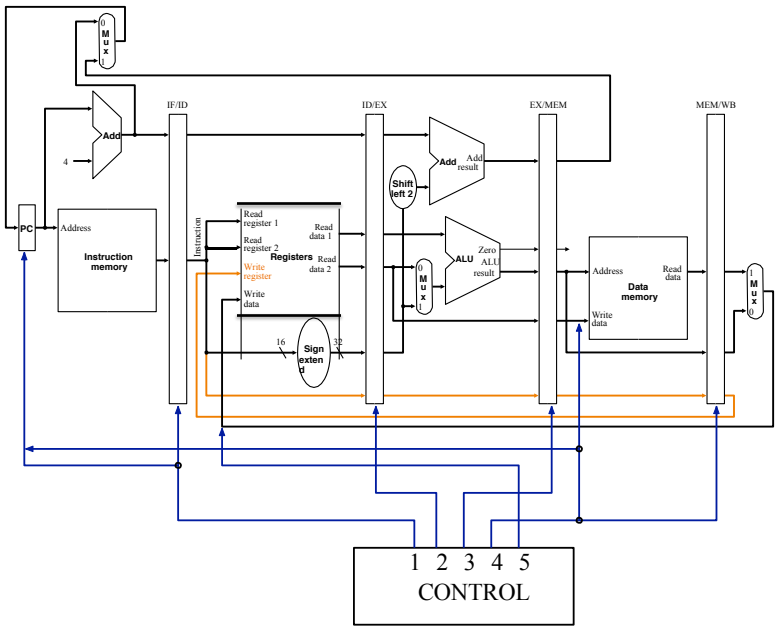
- NO READINGS : Multicycle Brief Overview
- Use several clock cycles(cc) to execute one instruction
- Break up original *long* cc into a *shorter* cc.
- At end of clock cycle, all data used in subsequent cycles must be stored in state element (intermediate registers)
- We assume one clock cycle can contain one memory access, a register file access (two reads or one write), or one ALU operation
- Once it has begun**, an instruction has exclusive use of the datapath until it completes**
- This datapath does <u>not</u> show implementation of control unit
- It is meant to be a conceptual overview of how to split up a datapath into stages. From *one* stage in Single Cycle to *five* stages in Multicycle.

Datapath for Multicycle Implementation

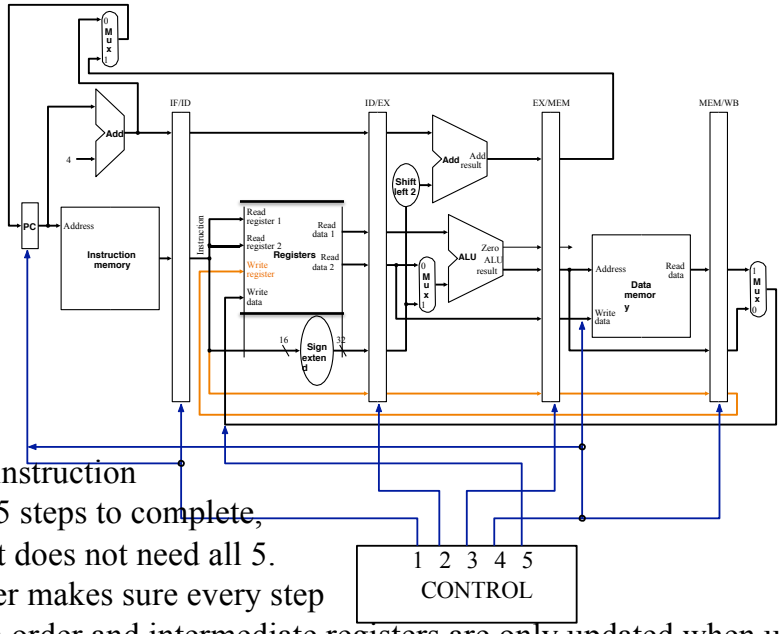


- This is Basic Single Cycle Datapath now with added intermediate registers
- Register files (IF/ID, etc) store information computed in one stage that's needed by later stages

Datapath/Control for Multicycle Implementation

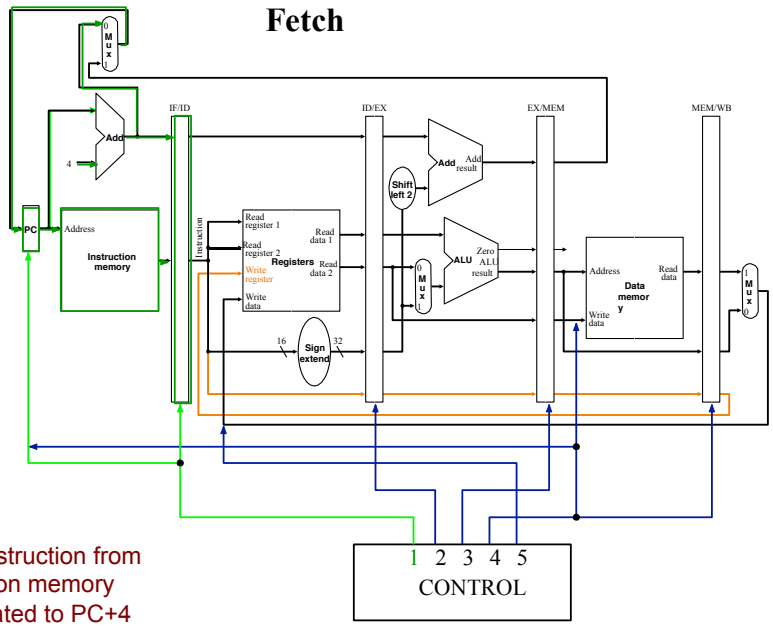


Datapath/Control for Multicycle Implementation



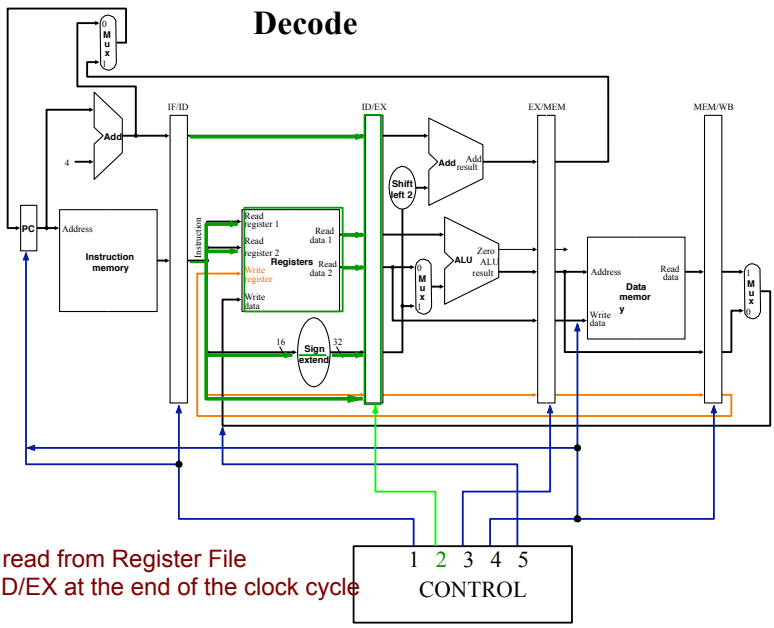
- Every instruction is given 5 steps to complete, Even if it does not need all 5. Controller makes sure every step occurs in order and intermediate registers are only updated when used.

Instruction Fetch



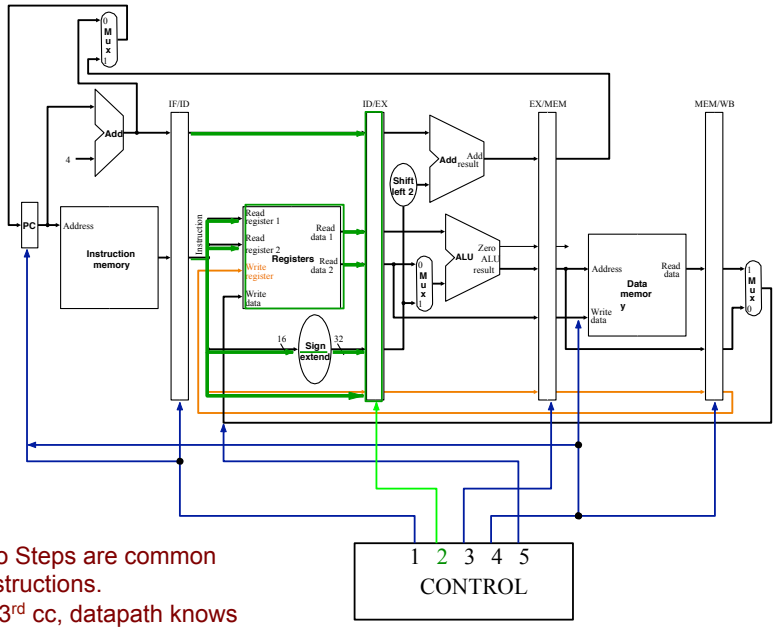
Fetch instruction from Instruction memory
PC updated to PC+4
End of this clock cycle : Write appropriate data to IF/ID

Instruction Decode



Perform read from Register File
Update ID/EX at the end of the clock cycle

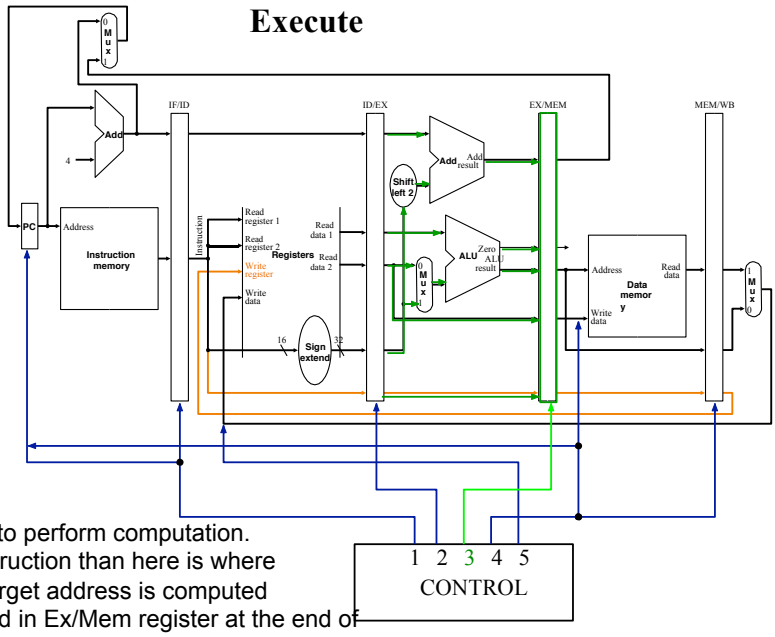
Instruction Decode



First Two Steps are common
To all instructions.
Now by 3rd cc, datapath knows
It is the **lw** instruction for example.

1 2 3 4 5
CONTROL

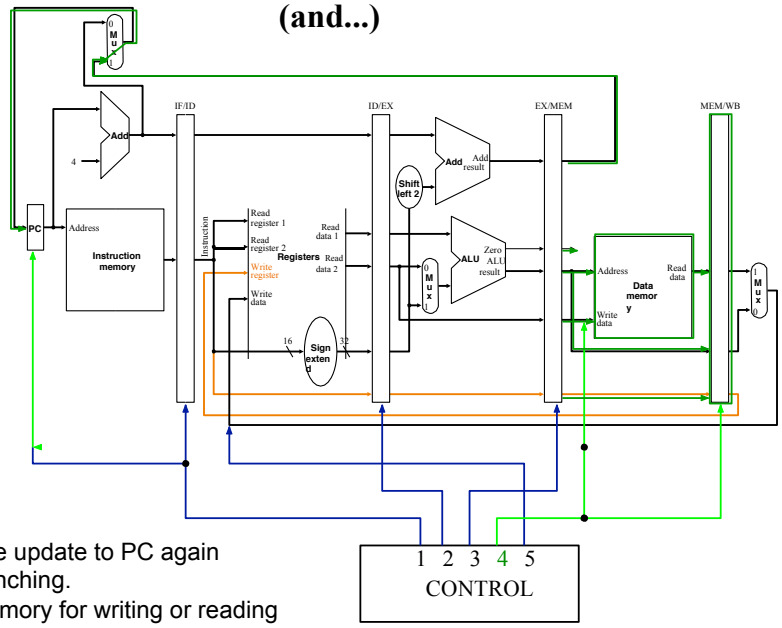
Instruction Execute



Use ALU to perform computation.
If beq instruction than here is where
Branch target address is computed
And stored in Ex/Mem register at the end of
This clock cycle.

1 2 3 4 5
CONTROL

Memory (and...)

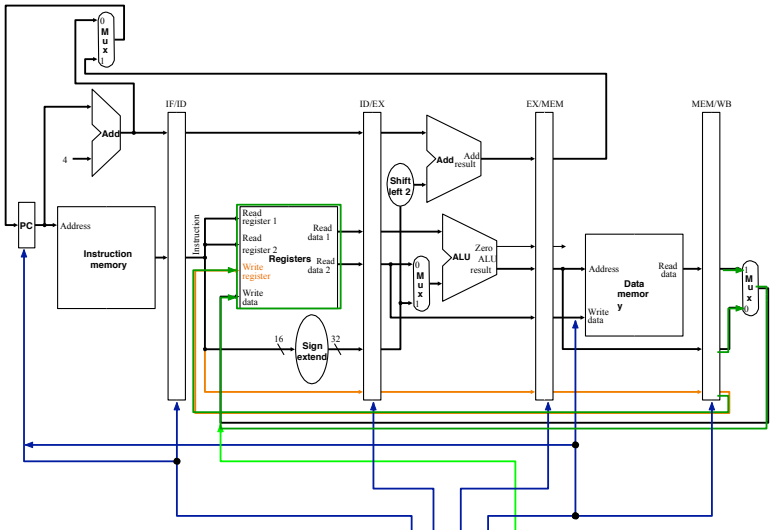


Possible update to PC again
For branching.
Use Memory for writing or reading

If we could have updated PC to branch target address sooner (ie step 03)
Would this have helped our Branch instruction's Total execution time on Multi-cycle datapath

a)No- each instruction takes 5 steps anyways
b)YES… we would fetch new instruction sooner
c)Sometimes

Write Back



This step occurs last and seems to be On the end of the datapath. However the work done is within the Datapath. Writing back to a register if necessary

Basic Control Unit

- Control steps through states 1,2,3,4,5 and repeats
- Can imagine smarter control:
  - beq and sw finish after state 4
  - j (not shown) finish after state 2

- We will see full smart control unit in the Pipelined Datapath

Performance

- Single cycle computer has 600ps clock
- Multicycle clock must be speed of *slowest* component.
  200ps (memory access)
- Each instruction takes 5 clock cycles
  Instructions use every stage, even some stages not needed
- Time per instruction: $5 \times 200\text{ps} =$
  1000ps Much slower than single cycle!
- Smarter control: 4 cc for quicker instructions : 800ps.

Notes on the Multicycle Datapath

- Real multicycle computer:
  - Much more hardware (stages)
    Not all stages used for all instructions
    Instructions only use stages they need
  - More complex instruction set
    Instructions will take different times to execute
  - Fast instructions take less time on multicycle since single cycle executes at speed of slowest
  - Control far more complex than ours (internal loops, branches)
    Finite State Machine
    Microprogramming
  - At any time, about 80% of hardware is **unused**