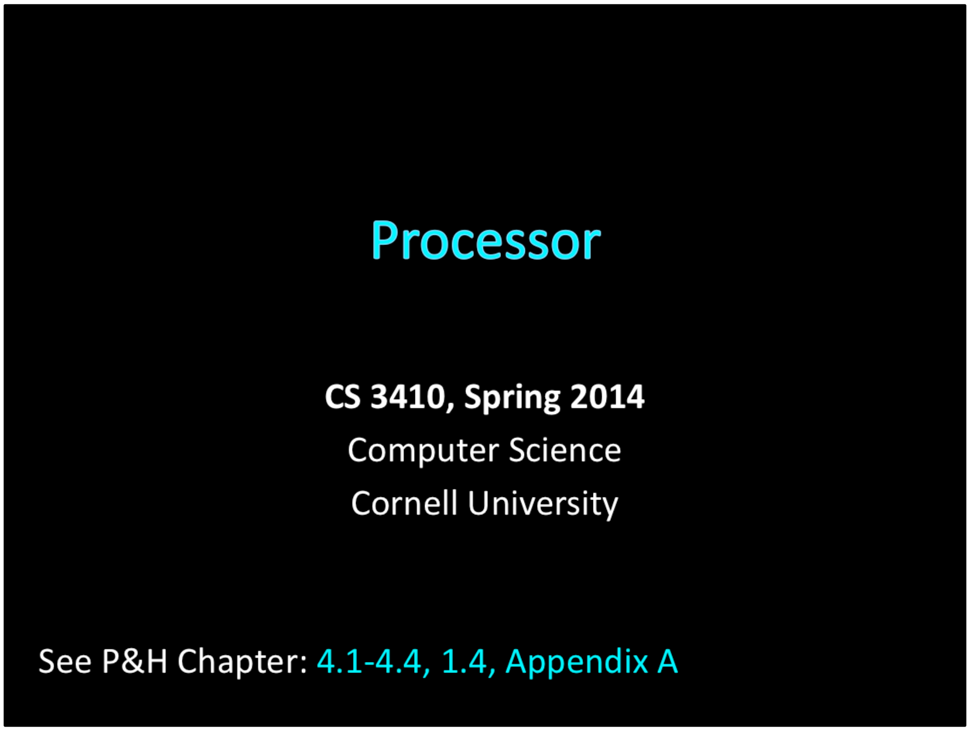
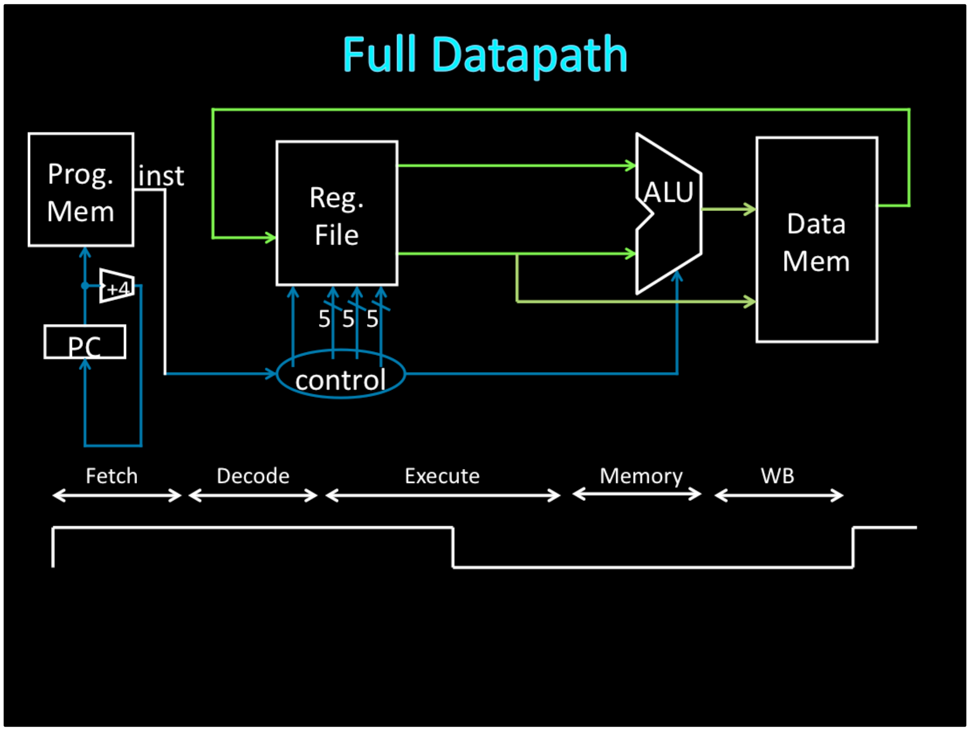


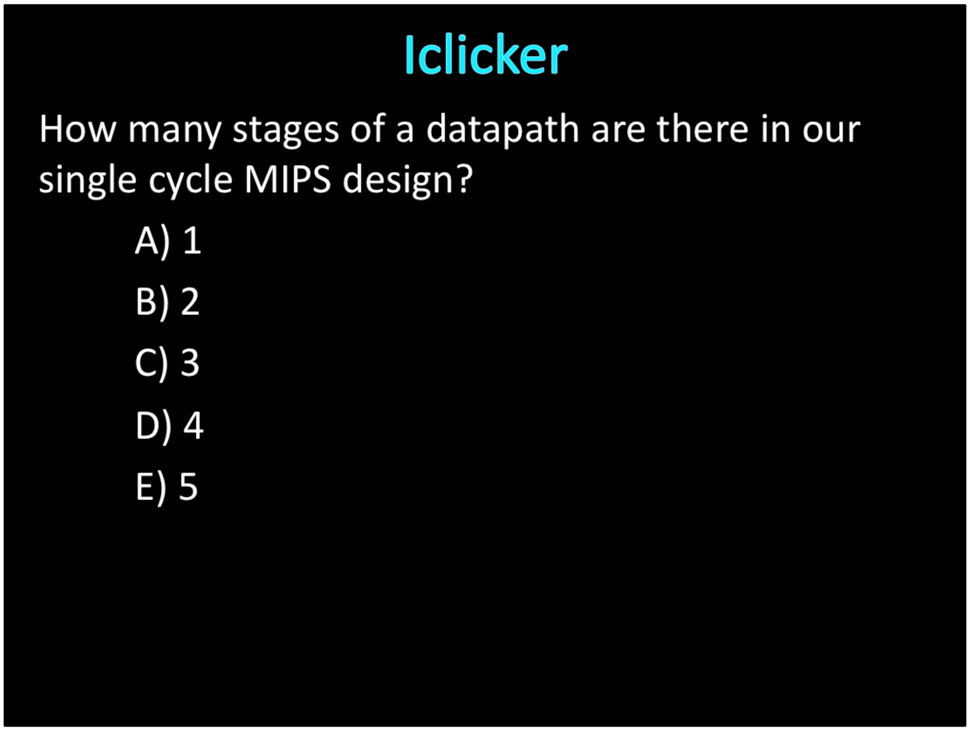
Processor

CS 3410, Spring 2014
Computer Science
Cornell University

See P&H Chapter: 4.1-4.4, 1.4, Appendix A



Full Datapath



Iclicker

How many stages of a datapath are there in our single cycle MIPS design?

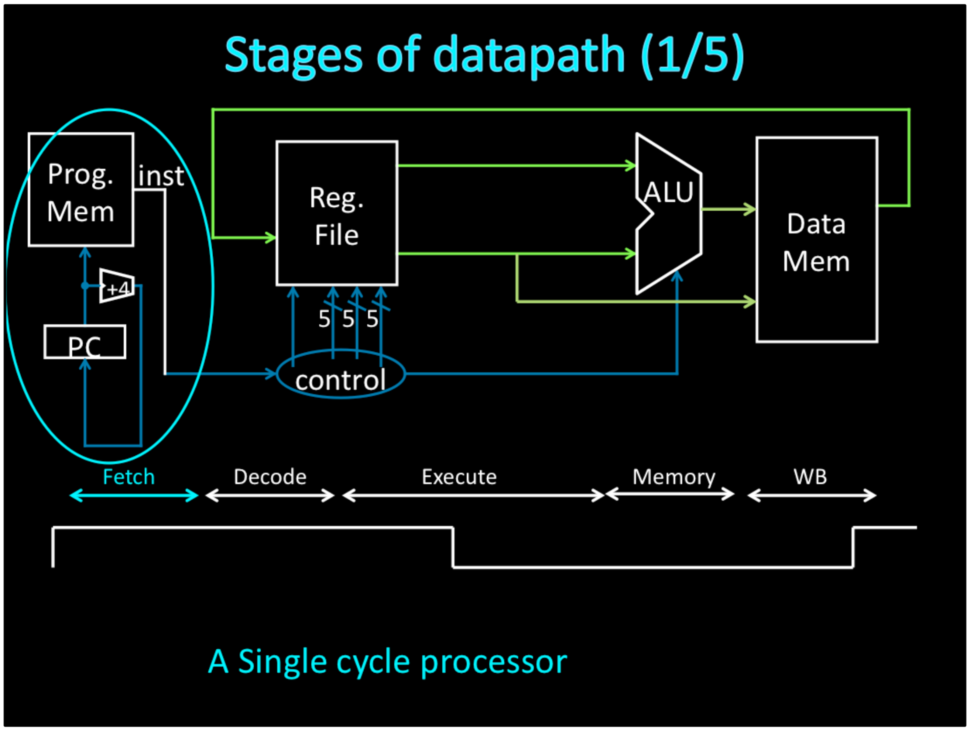
A) 1

B) 2

C) 3

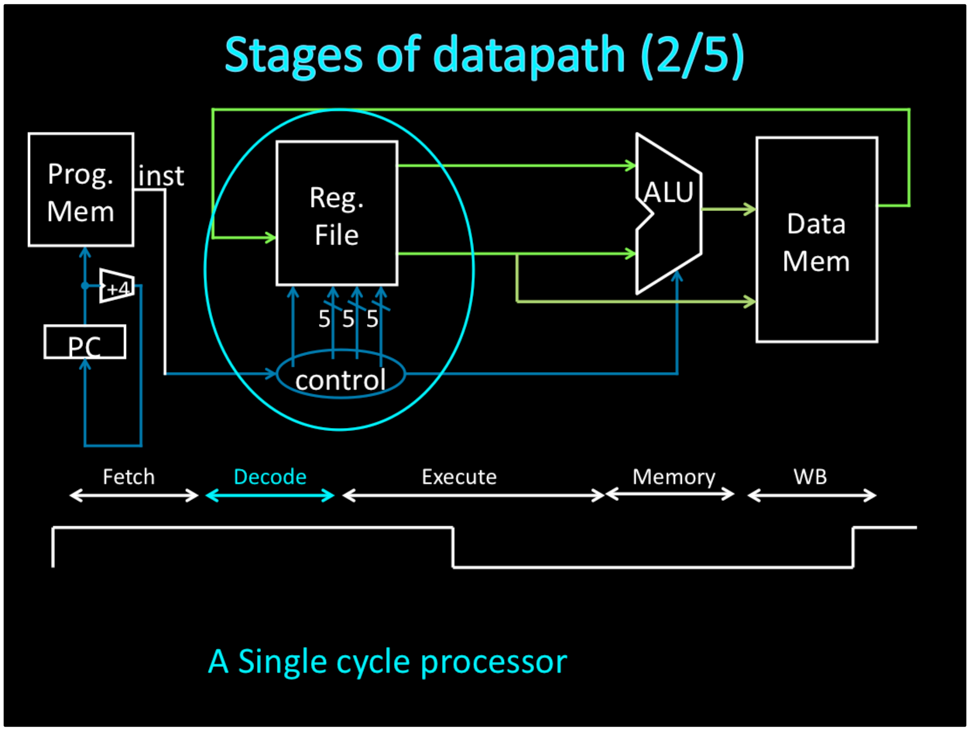
D) 4

E) 5



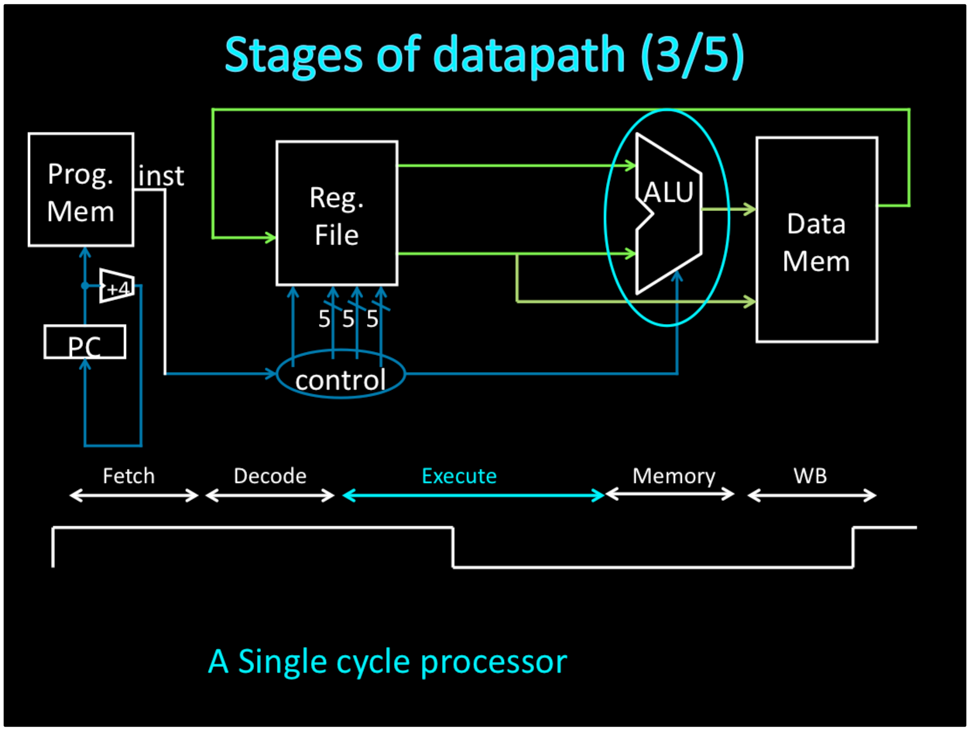
Stages of datapath (1/5)

A Single cycle processor



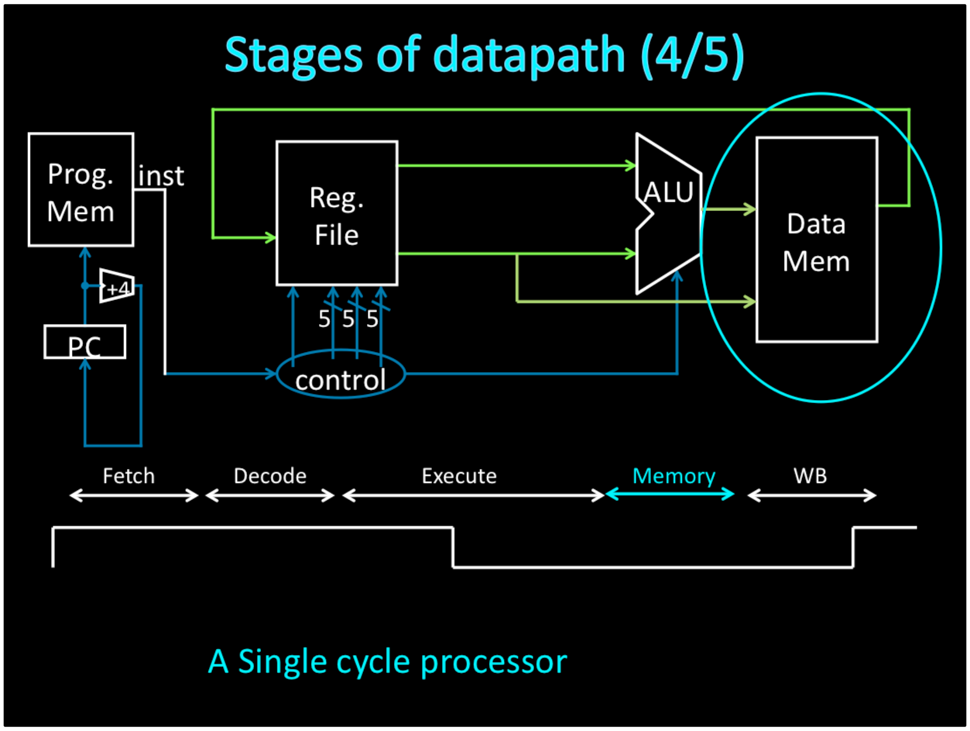
Stages of datapath (2/5)

A Single cycle processor



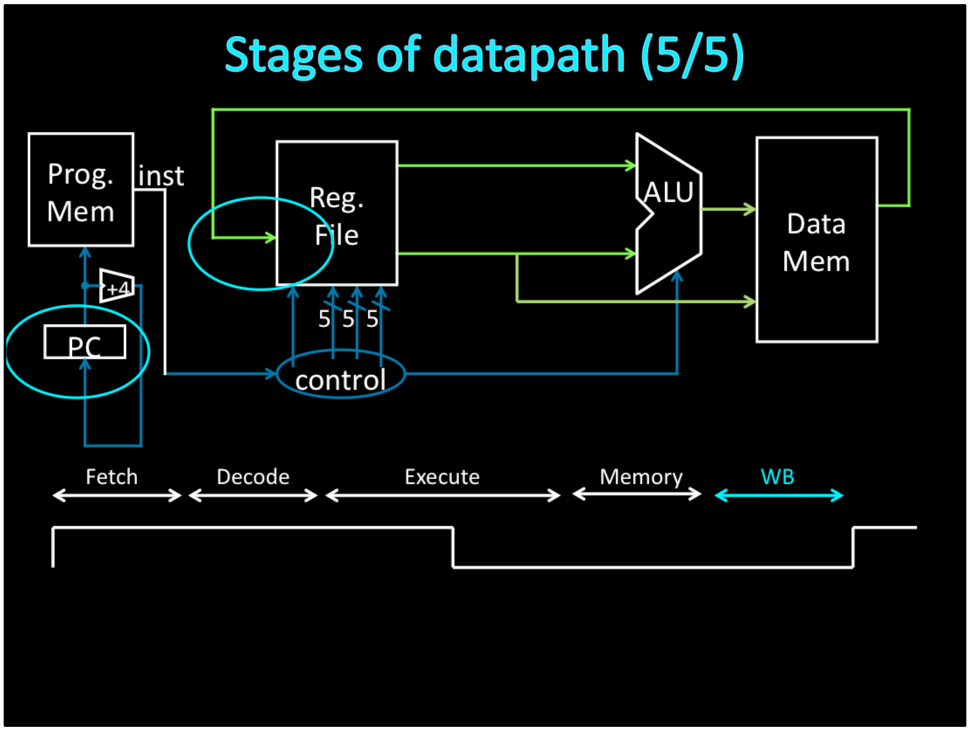
Stages of datapath (3/5)

A Single cycle processor

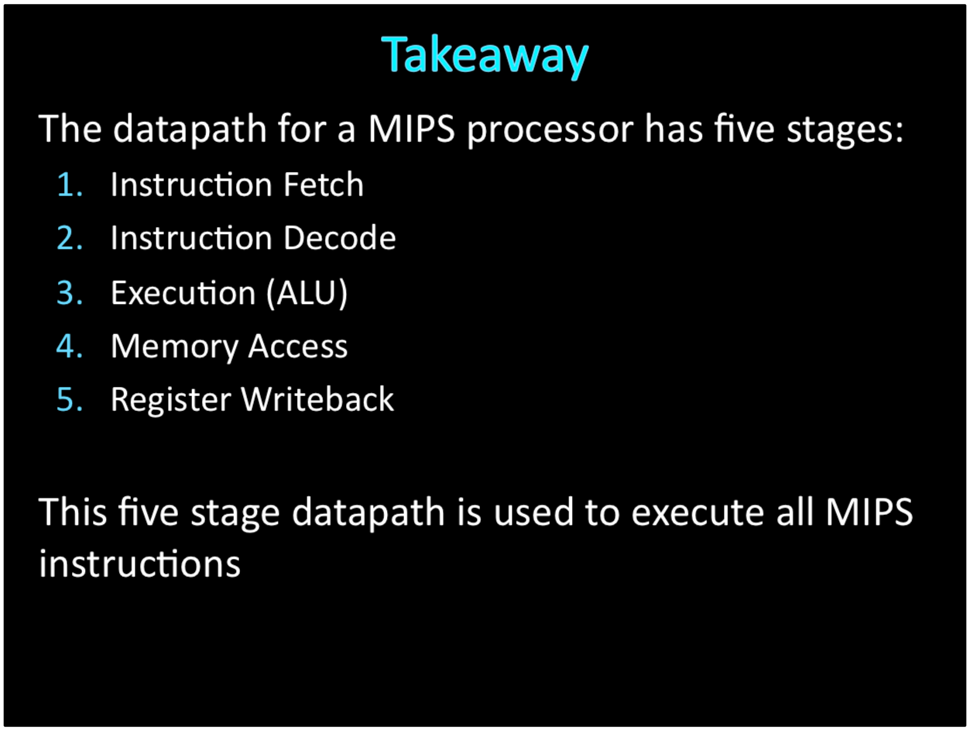


Stages of datapath (4/5)

A Single cycle processor



Stages of datapath (5/5)

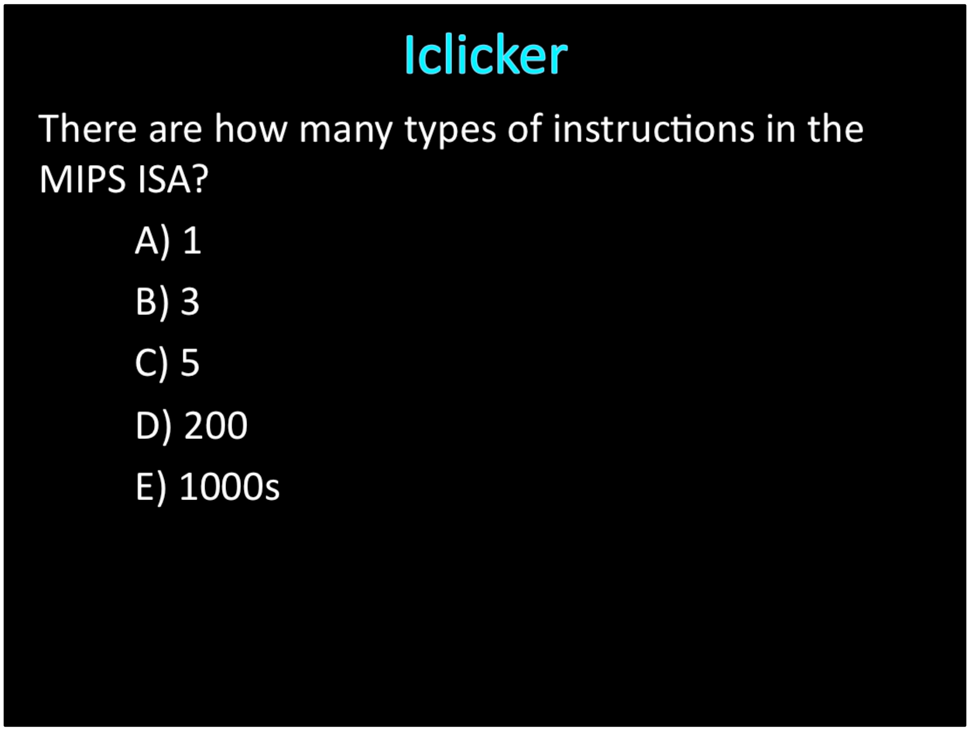


Takeaway

The datapath for a MIPS processor has five stages:

1. Instruction Fetch
2. Instruction Decode
3. Execution (ALU)
4. Memory Access
5. Register Writeback

This five stage datapath is used to execute all MIPS instructions



Iclicker

There are how many types of instructions in the MIPS ISA?

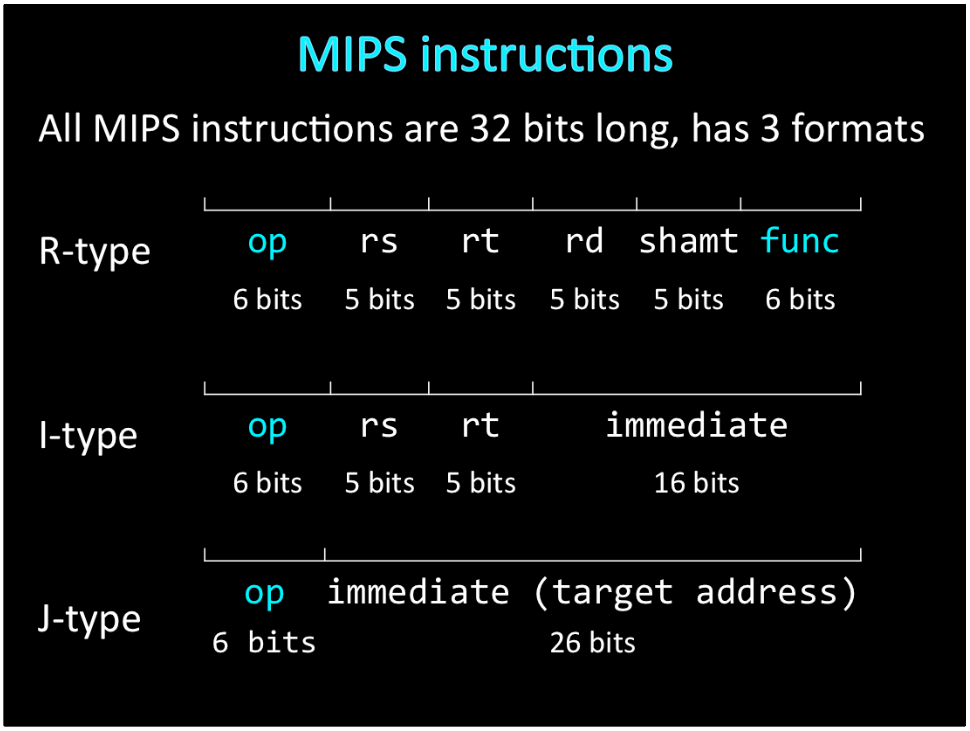
A) 1

B) 3

C) 5

D) 200

E) 1000s



MIPS instructions

All MIPS instructions are 32 bits long, has 3 formats

R-type	op	rs	rt	rd	shamt	func
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

I-type	op	rs	rt	immediate
	6 bits	5 bits	5 bits	16 bits

J-type	op	immediate (target address)
	6 bits	26 bits

MIPS Instruction Functions

Arithmetic/Logical

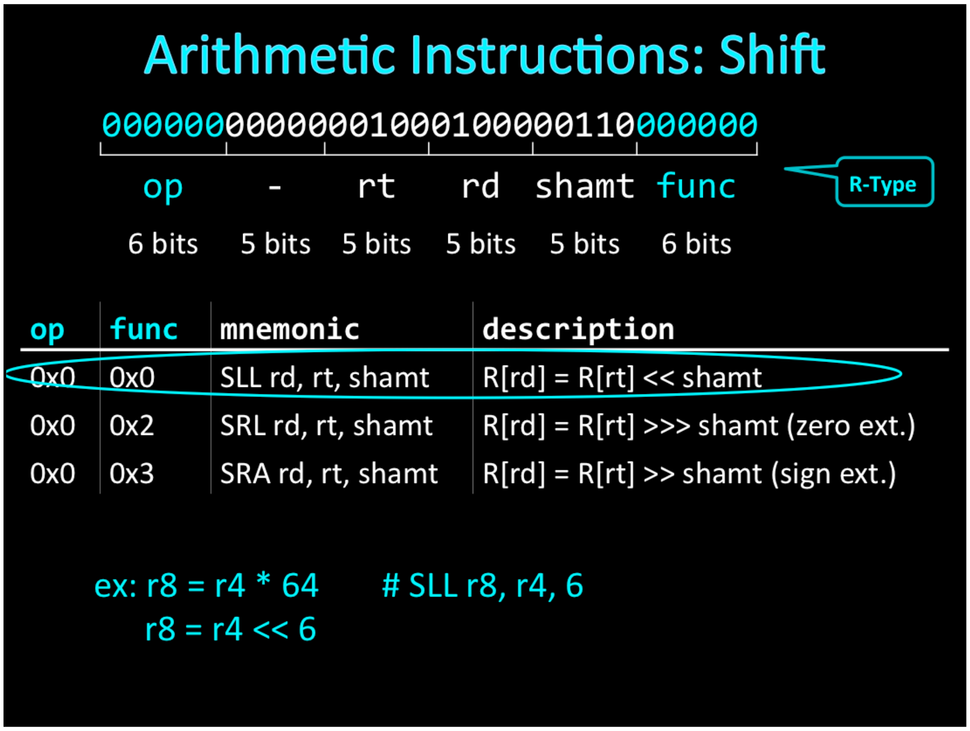
- R-type: result and two source registers, shift amount
- I-type: 16-bit immediate with sign/zero extension

Memory Access

- load/store between registers and memory
- word, half-word and byte operations

Control flow

- conditional branches: pc-relative addresses
- jumps: fixed offsets, register absolute



Arithmetic Instructions: Shift

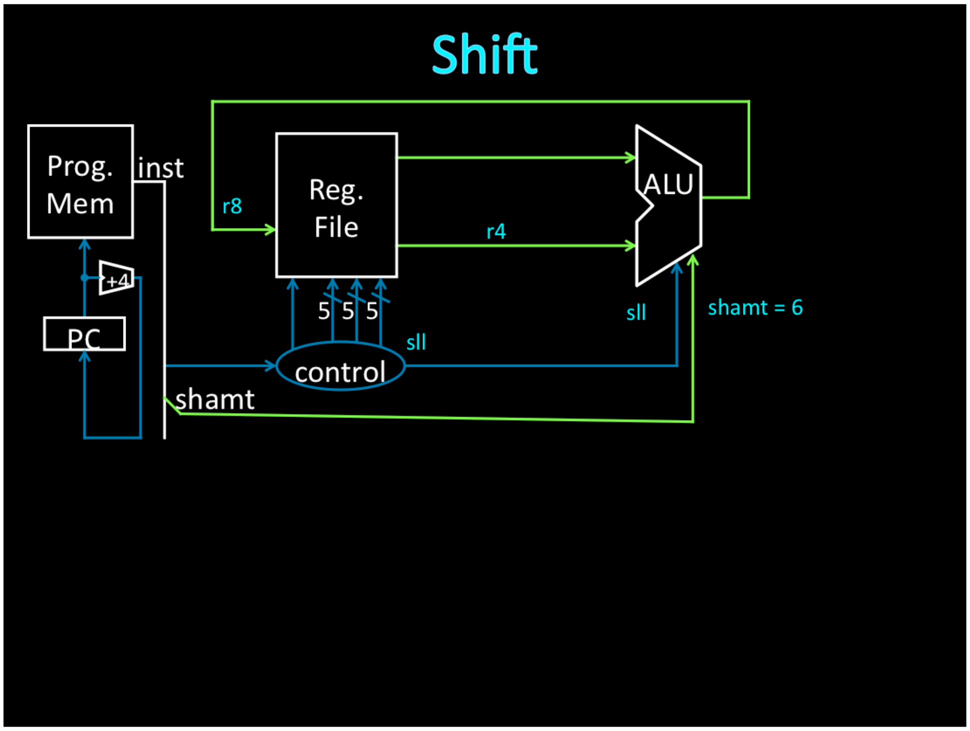
```
00000000000000010001000001100000000
```

op	-	rt	rd	shamt	func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

R-Type

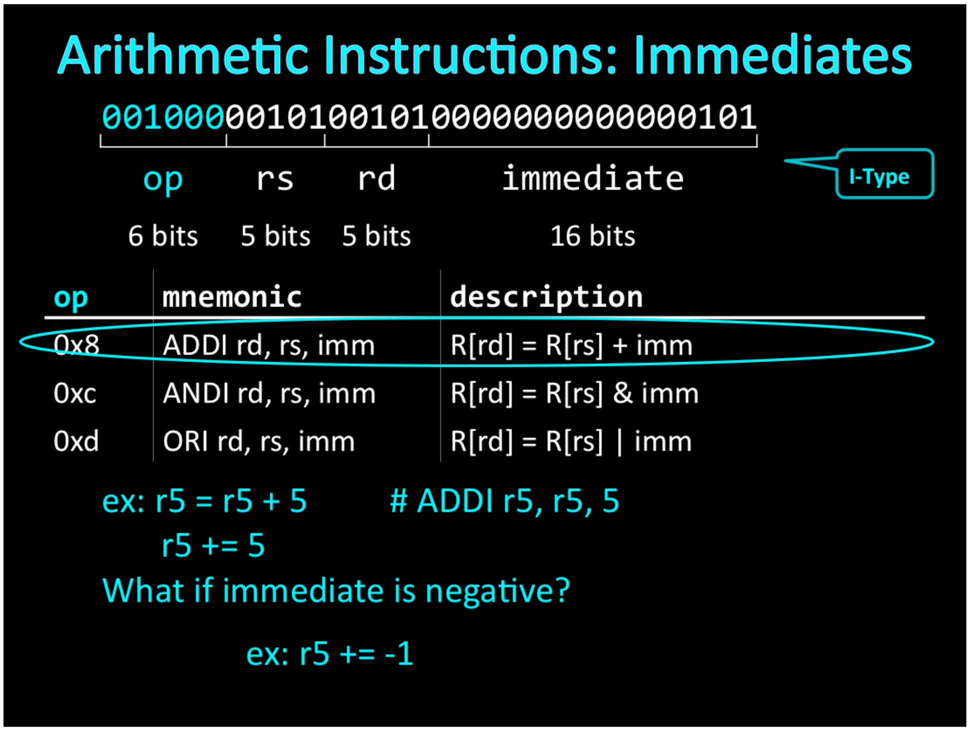
op	func	mnemonic	description
0x0	0x0	SLL rd, rt, shamt	R[rd] = R[rt] << shamt
0x0	0x2	SRL rd, rt, shamt	R[rd] = R[rt] >>> shamt (zero ext.)
0x0	0x3	SRA rd, rt, shamt	R[rd] = R[rt] >> shamt (sign ext.)

ex: r8 = r4 * 64 # SLL r8, r4, 6
r8 = r4 << 6



Shift

Prog.
Mem
inst
Reg.
File
ALU
r8
r4
+4
PC
5
5
5
control
sll
sll
shamt
shamt = 6



Arithmetic Instructions: Immediates

`00100000101001010000000000000101`

op	rs	rd	immediate
6 bits	5 bits	5 bits	16 bits

I-Type

op	mnemonic	description
0x8	ADDI rd, rs, imm	R[rd] = R[rs] + imm
0xc	ANDI rd, rs, imm	R[rd] = R[rs] & imm
0xd	ORI rd, rs, imm	R[rd] = R[rs] | imm

ex: r5 = r5 + 5 # ADDI r5, r5, 5

r5 += 5

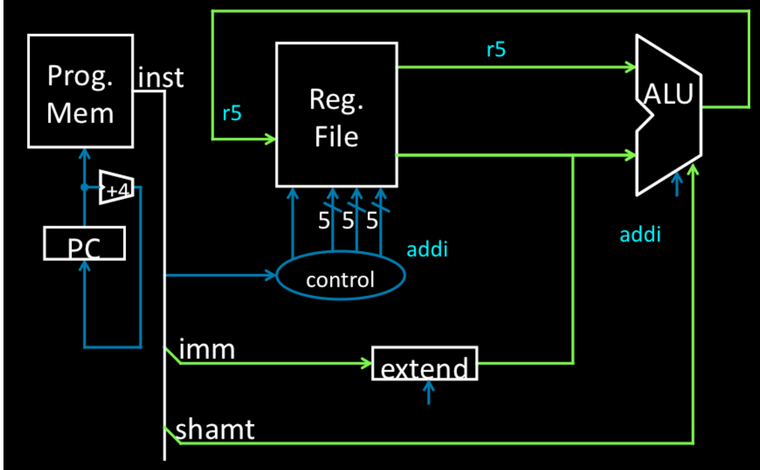
What if immediate is negative?

ex: r5 += -1

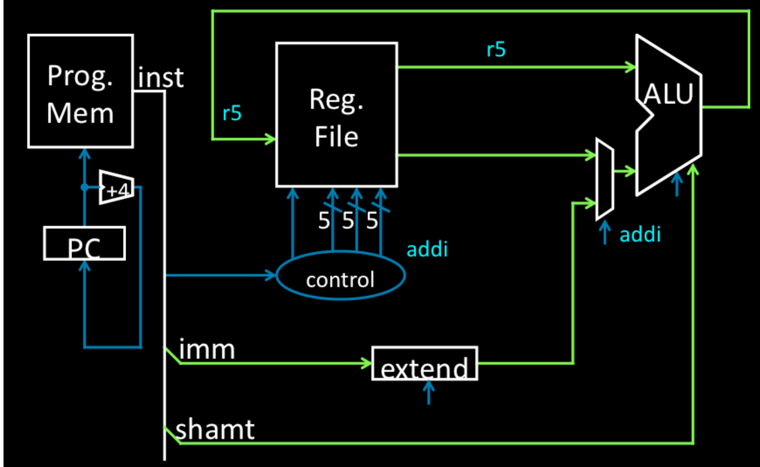
For addi, sign extension and can do positive and negative. Will signal overflow.
For addiu, still use sign extension, but no overflow

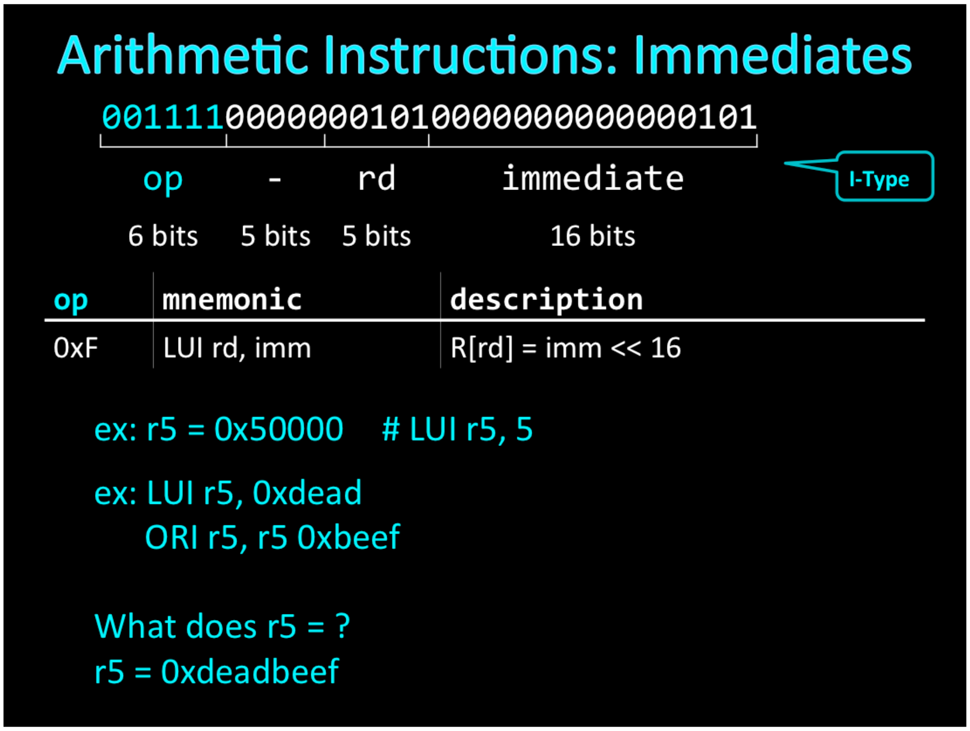
Add vs. addu: no overflow for addu

Immediates


Prog.
Mem
inst
+4
PC
r5
Reg.
File
r5
ALU
5
5
5
addi
control
addi
imm
extend
shamt

Immediates


Prog.
Mem
inst
r5
Reg.
File
r5
ALU
+4
PC
5
5
5
addi
control
addi
imm
extend
shamt



Arithmetic Instructions: Immediates

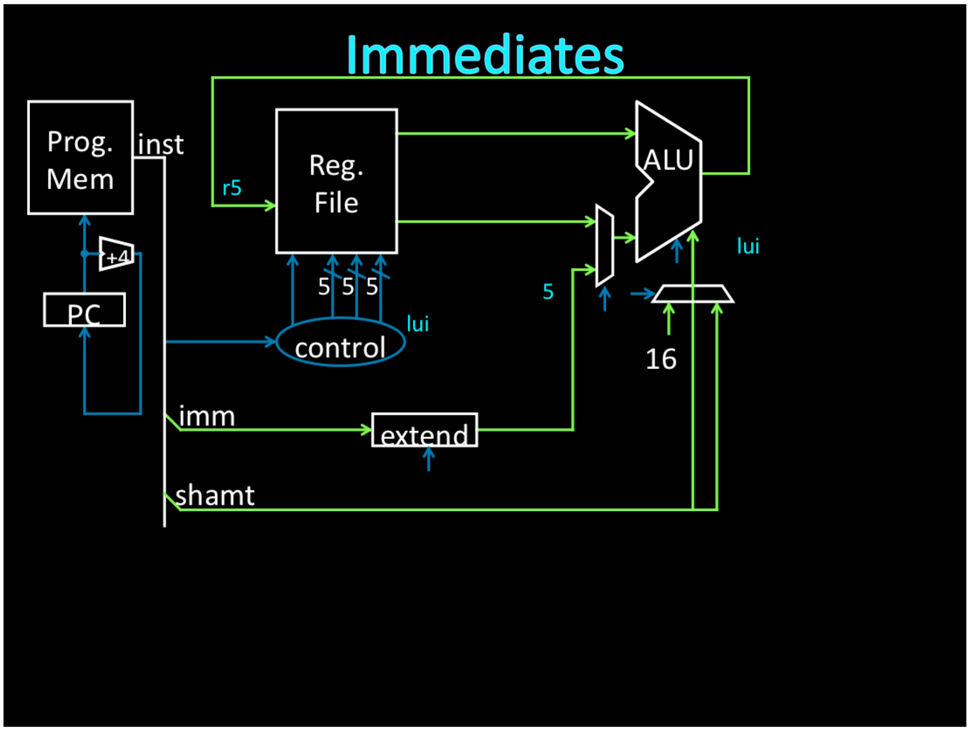
```
00111100000001010000000000000101
  op      -      rd      immediate
6 bits  5 bits  5 bits   16 bits
```

op	mnemonic	description
0xF	LUI rd, imm	R[rd] = imm << 16

ex: r5 = 0x50000   # LUI r5, 5

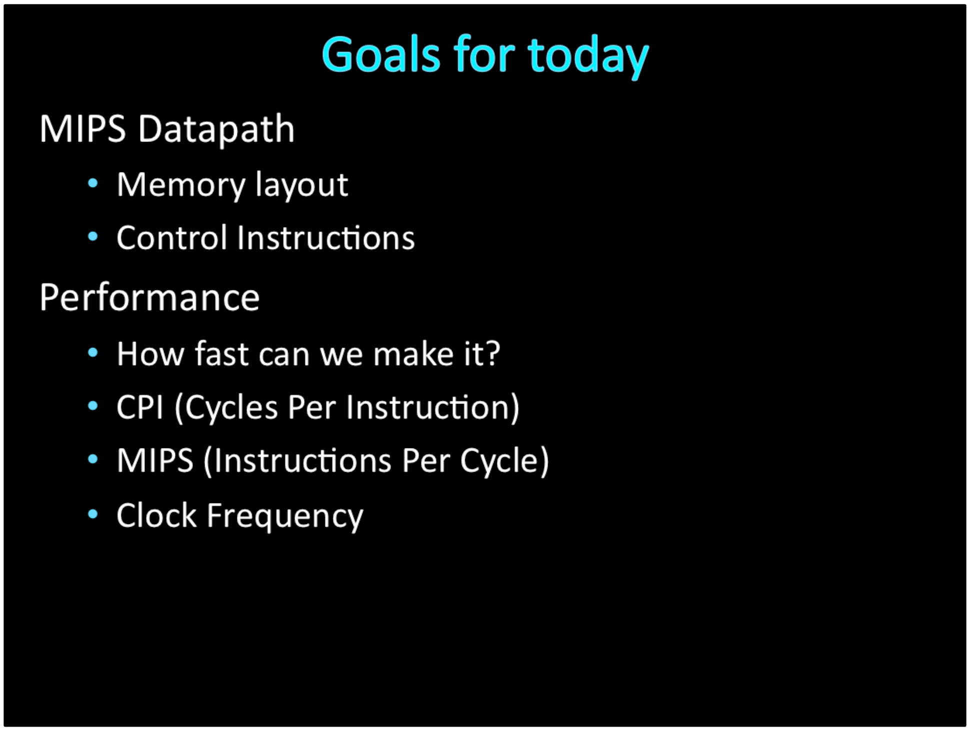
ex: LUI r5, 0xdead
    ORI r5, r5 0xbeef

What does r5 = ?
r5 = 0xdeadbeef



Immediates

or, extra input to alu B mux from before/after extend
(or, extra mux after alu)



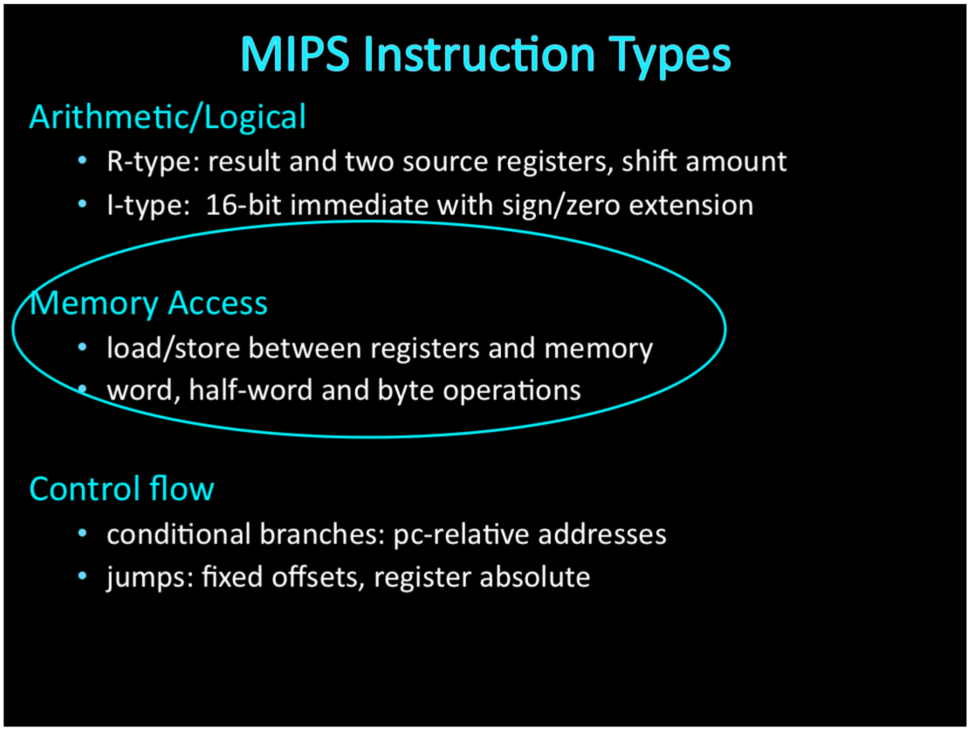
Goals for today

MIPS Datapath

- Memory layout
- Control Instructions

Performance

- How fast can we make it?
- CPI (Cycles Per Instruction)
- MIPS (Instructions Per Cycle)
- Clock Frequency



MIPS Instruction Types

Arithmetic/Logical

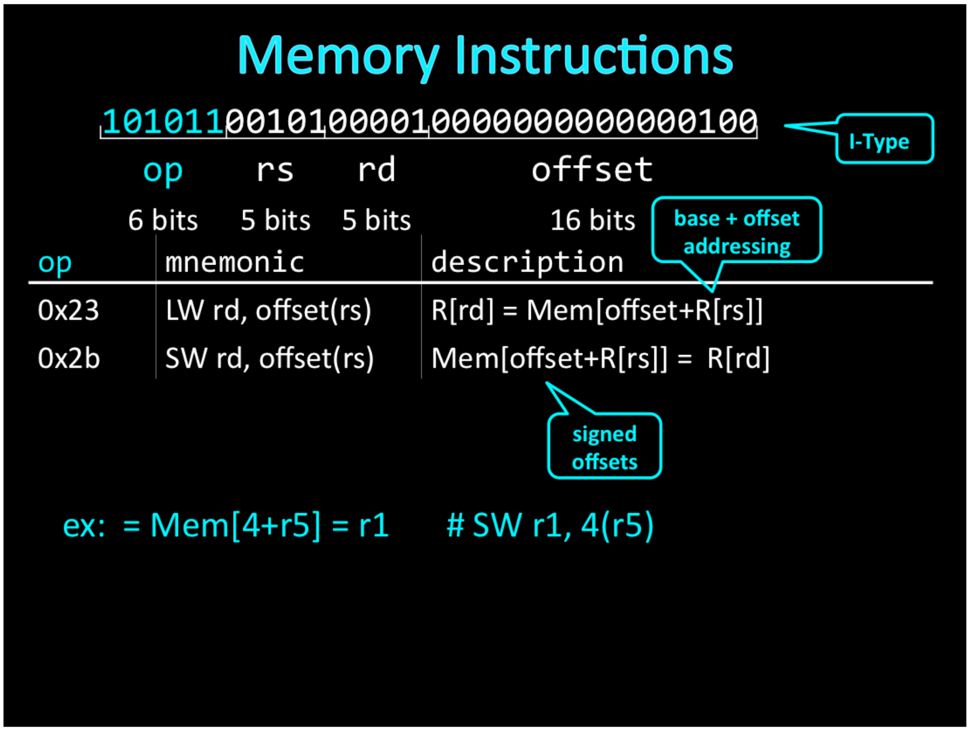
- R-type: result and two source registers, shift amount
- I-type: 16-bit immediate with sign/zero extension

Memory Access

- load/store between registers and memory
- word, half-word and byte operations

Control flow

- conditional branches: pc-relative addresses
- jumps: fixed offsets, register absolute



Memory Instructions

```
10101100101000010000000000000100
  op     rs    rd       offset
 6 bits 5 bits 5 bits   16 bits
```

I-Type

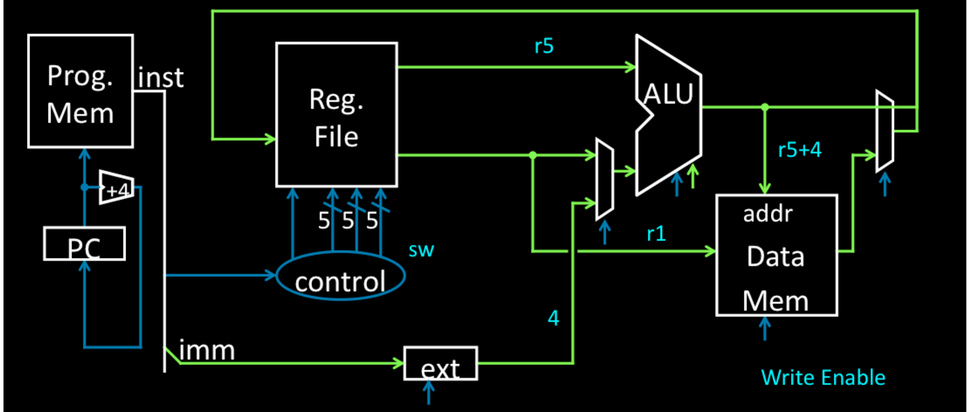
base + offset addressing

op	mnemonic	description
0x23	LW rd, offset(rs)	R[rd] = Mem[offset+R[rs]]
0x2b	SW rd, offset(rs)	Mem[offset+R[rs]] = R[rd]

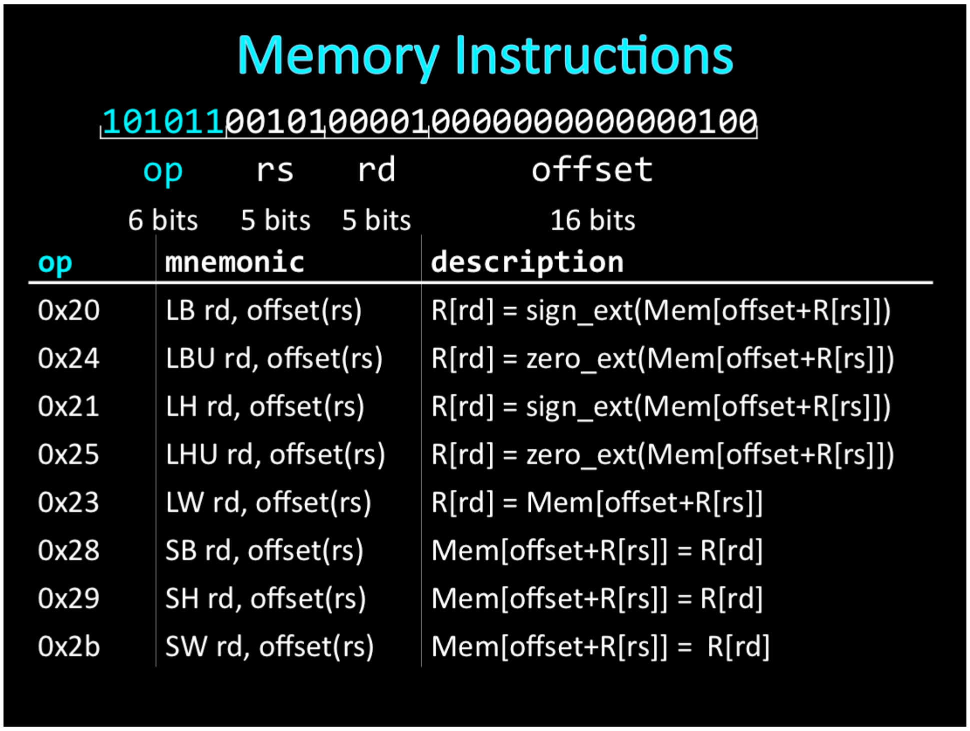
signed offsets

ex: = Mem[4+r5] = r1 # SW r1, 4(r5)

Memory Operations



ex: = Mem[4+r5] = r1 # SW r1, 4(r5)



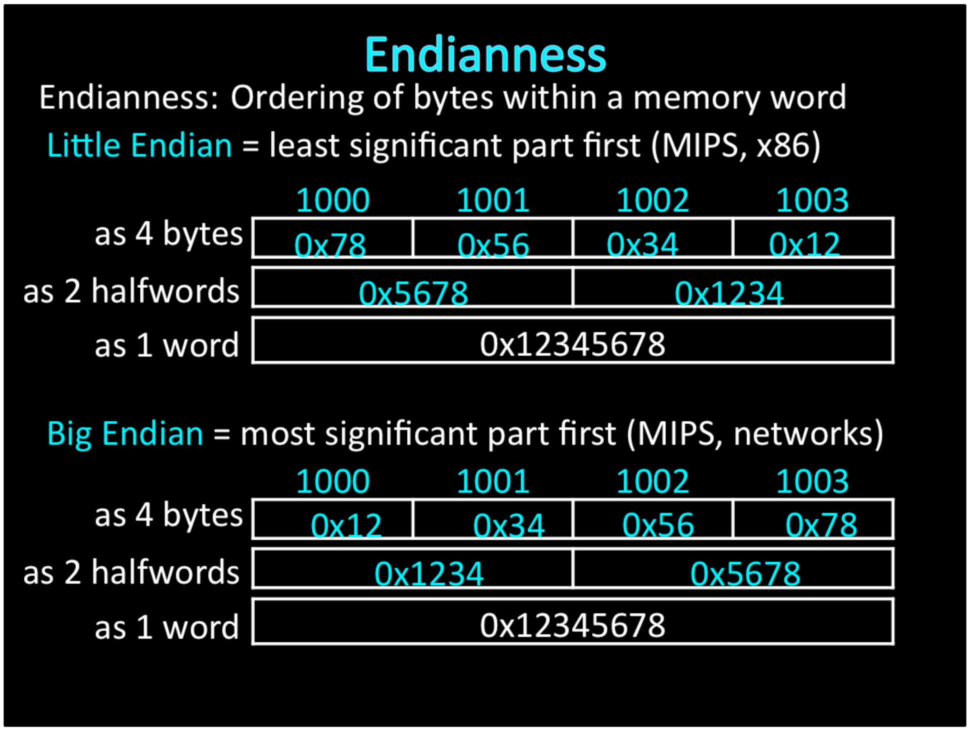
Memory Instructions

101011	00101	00001	0000000000000100
op	rs	rd	offset
6 bits	5 bits	5 bits	16 bits

op	mnemonic	description
0x20	LB rd, offset(rs)	R[rd] = sign_ext(Mem[offset+R[rs]])
0x24	LBU rd, offset(rs)	R[rd] = zero_ext(Mem[offset+R[rs]])
0x21	LH rd, offset(rs)	R[rd] = sign_ext(Mem[offset+R[rs]])
0x25	LHU rd, offset(rs)	R[rd] = zero_ext(Mem[offset+R[rs]])
0x23	LW rd, offset(rs)	R[rd] = Mem[offset+R[rs]]
0x28	SB rd, offset(rs)	Mem[offset+R[rs]] = R[rd]
0x29	SH rd, offset(rs)	Mem[offset+R[rs]] = R[rd]
0x2b	SW rd, offset(rs)	Mem[offset+R[rs]] = R[rd]

sw r1 4(r5)
sb r1, 3(r5)

If you don’t make sure half word accesses are half word aligned, or word accesses are word aligned, there will be an error signaled. We will talk about traps and exceptions later.



Endianness

Endianness: Ordering of bytes within a memory word

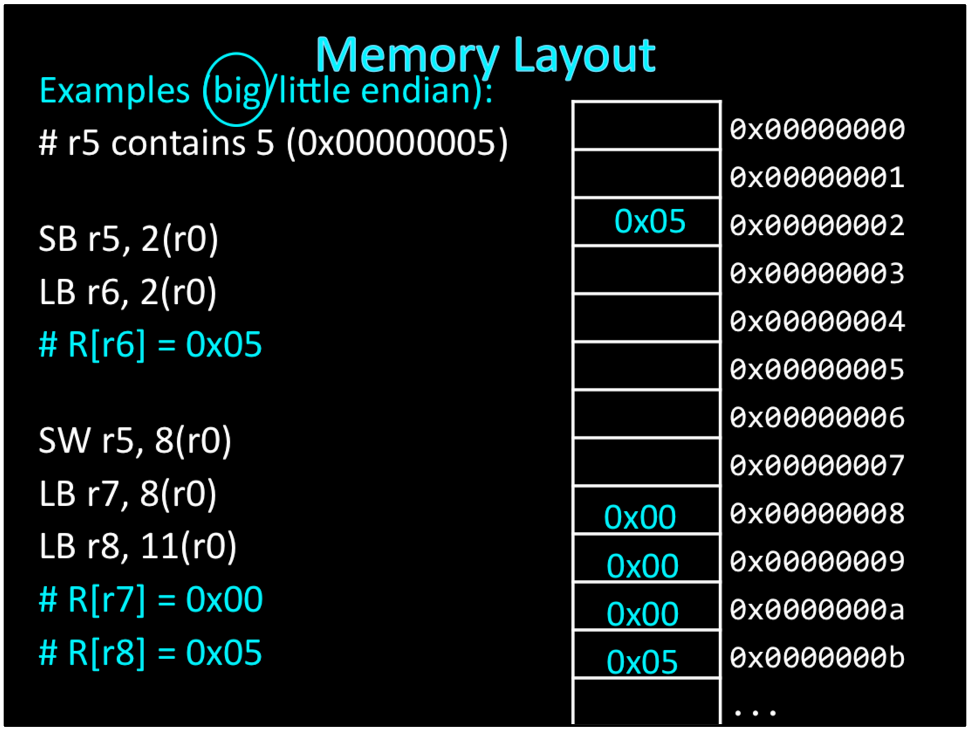
Little Endian = least significant part first (MIPS, x86)

	1000	1001	1002	1003
as 4 bytes	0x78	0x56	0x34	0x12
as 2 halfwords	0x5678		0x1234	
as 1 word	0x12345678			

Big Endian = most significant part first (MIPS, networks)

	1000	1001	1002	1003
as 4 bytes	0x12	0x34	0x56	0x78
as 2 halfwords	0x1234		0x5678	
as 1 word	0x12345678			

Comes from Gulliver travels



Memory Layout

Examples (big/little endian):

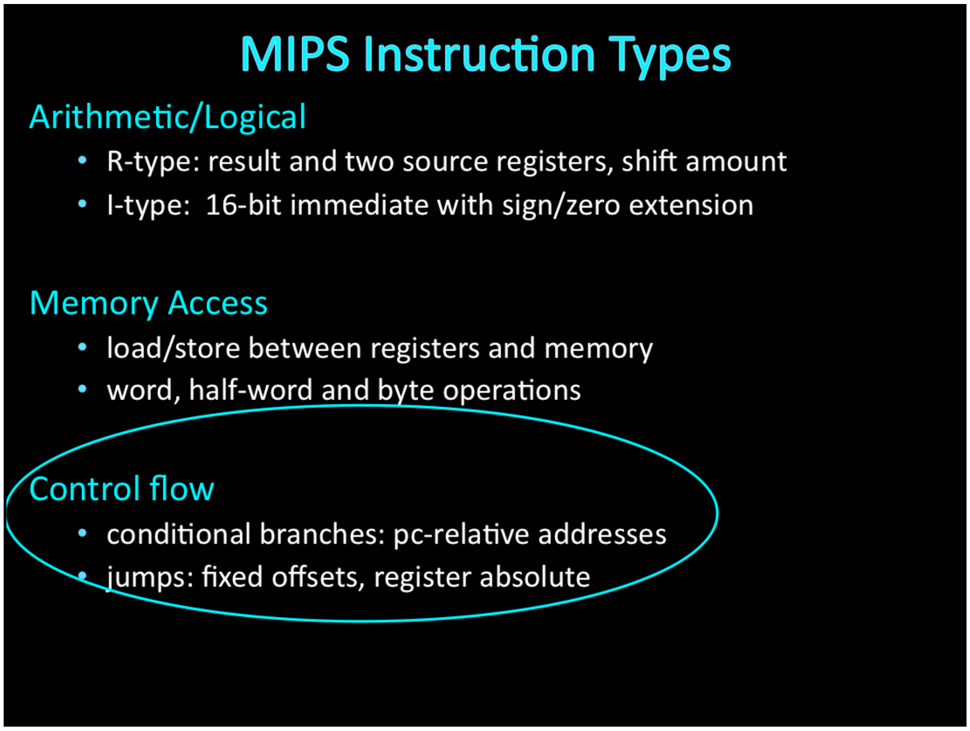
```
# r5 contains 5 (0x00000005)

SB r5, 2(r0)
LB r6, 2(r0)
# R[r6] = 0x05

SW r5, 8(r0)
LB r7, 8(r0)
LB r8, 11(r0)
# R[r7] = 0x00
# R[r8] = 0x05
```

	0x00000000
	0x00000001
0x05	0x00000002
	0x00000003
	0x00000004
	0x00000005
	0x00000006
	0x00000007
0x00	0x00000008
0x00	0x00000009
0x00	0x0000000a
0x05	0x0000000b
	...

Big Endian means store MSB (most significant byte) first



MIPS Instruction Types

Arithmetic/Logical

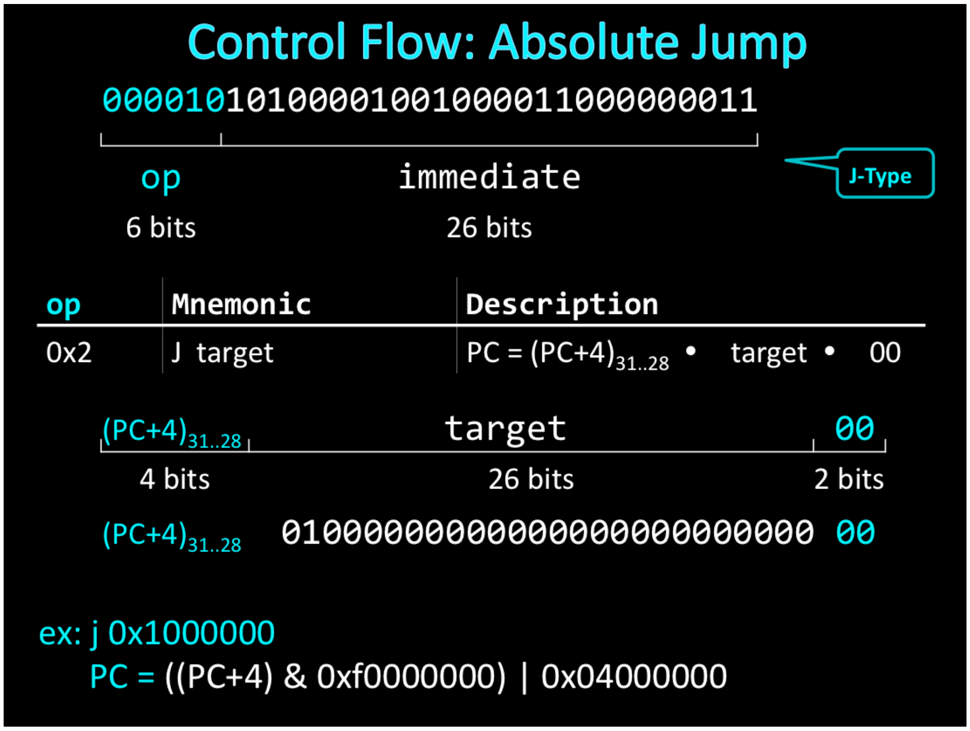
- R-type: result and two source registers, shift amount
- I-type: 16-bit immediate with sign/zero extension

Memory Access

- load/store between registers and memory
- word, half-word and byte operations

Control flow

- conditional branches: pc-relative addresses
- jumps: fixed offsets, register absolute



Control Flow: Absolute Jump

op	Mnemonic	Description
0x2	J target	PC = $(PC+4)_{31..28}$ • target • 00

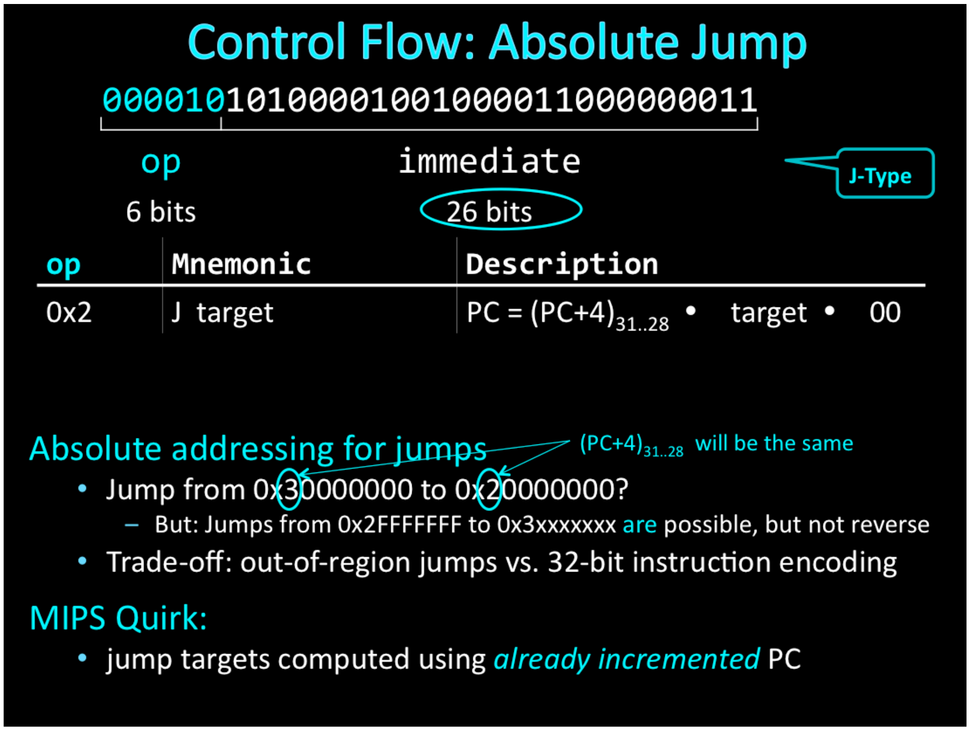
ex: j 0x1000000

PC = ((PC+4) & 0xf0000000) | 0x04000000

Where • is used to concatenate

Why should the offset be left shifted by 2. To keep the jump address word aligned.

AND to not waste two bits that we *know* are going to be 0.



Control Flow: Absolute Jump

```
00001010100001001000011000000011
```

$\underbrace{000010}_{\text{op}}\underbrace{10100001001000011000000011}_{\text{immediate}}$

op — 6 bits

immediate — 26 bits

J-Type

op	Mnemonic	Description
0x2	J target	PC = $(PC+4)_{31..28}$ • target • 00

Absolute addressing for jumps

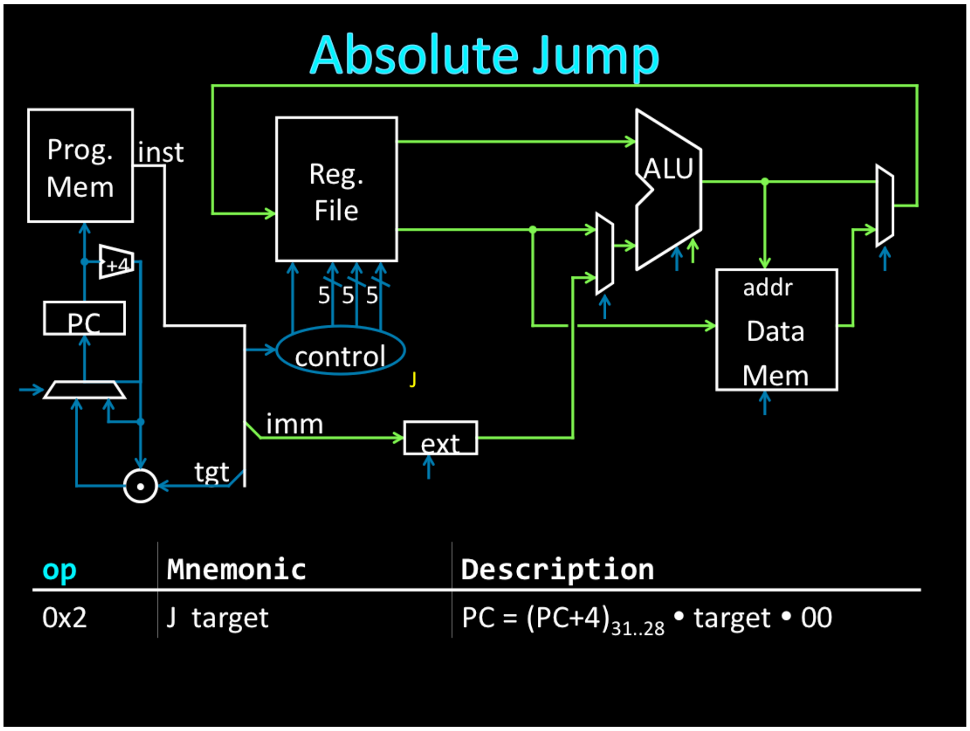
$(PC+4)_{31..28}$ will be the same

- Jump from 0x30000000 to 0x20000000?
  - But: Jumps from 0x2FFFFFFF to 0x3xxxxxxx are possible, but not reverse
- Trade-off: out-of-region jumps vs. 32-bit instruction encoding

MIPS Quirk:

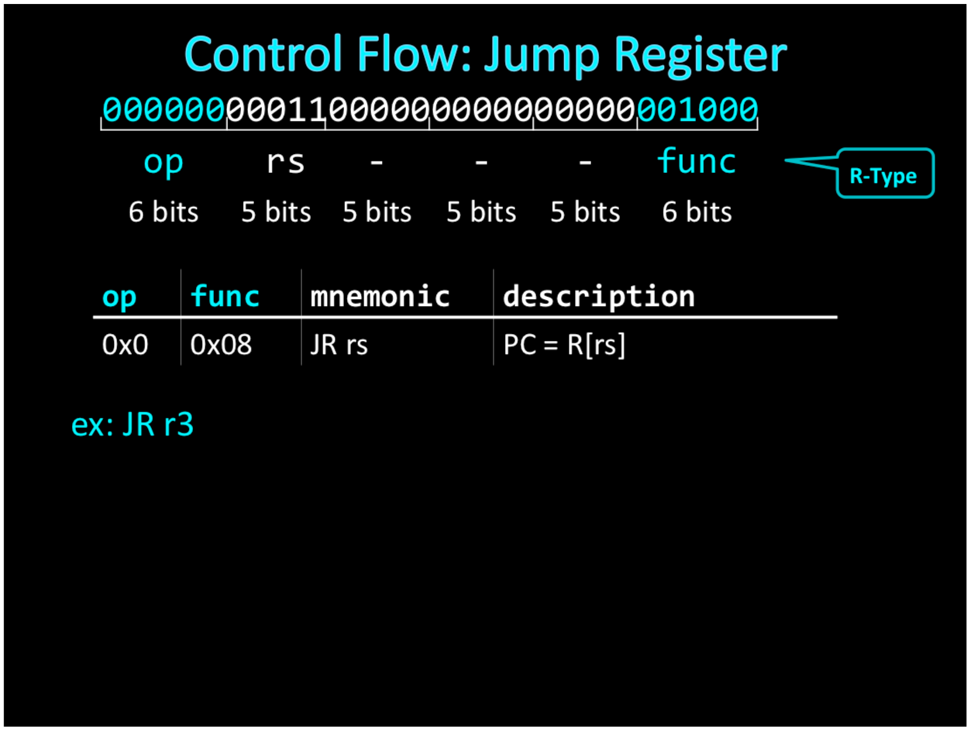
- jump targets computed using *already incremented* PC

Where • is used to concatenate



Absolute Jump

op	Mnemonic	Description
0x2	J target	$PC = (PC+4)_{31..28} \bullet \text{target} \bullet 00$



Control Flow: Jump Register

```
00000000011000000000000000001000
```

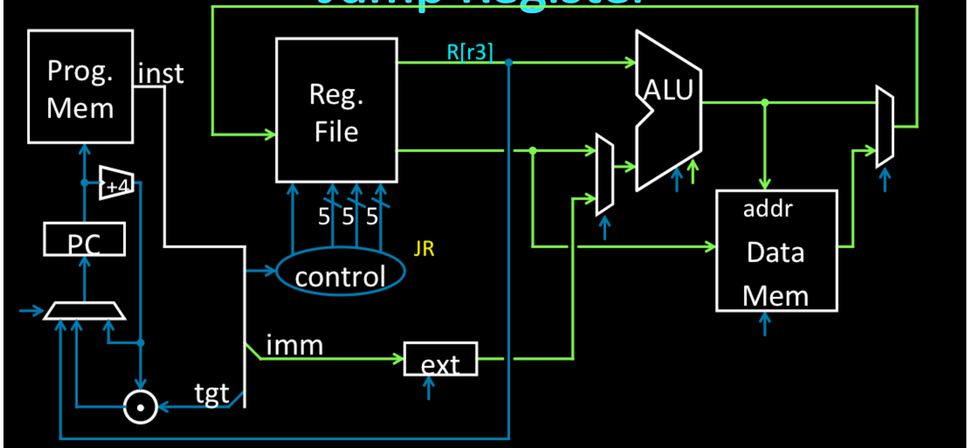
000000	00011	00000	00000	00000	001000
op	rs	-	-	-	func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

R-Type

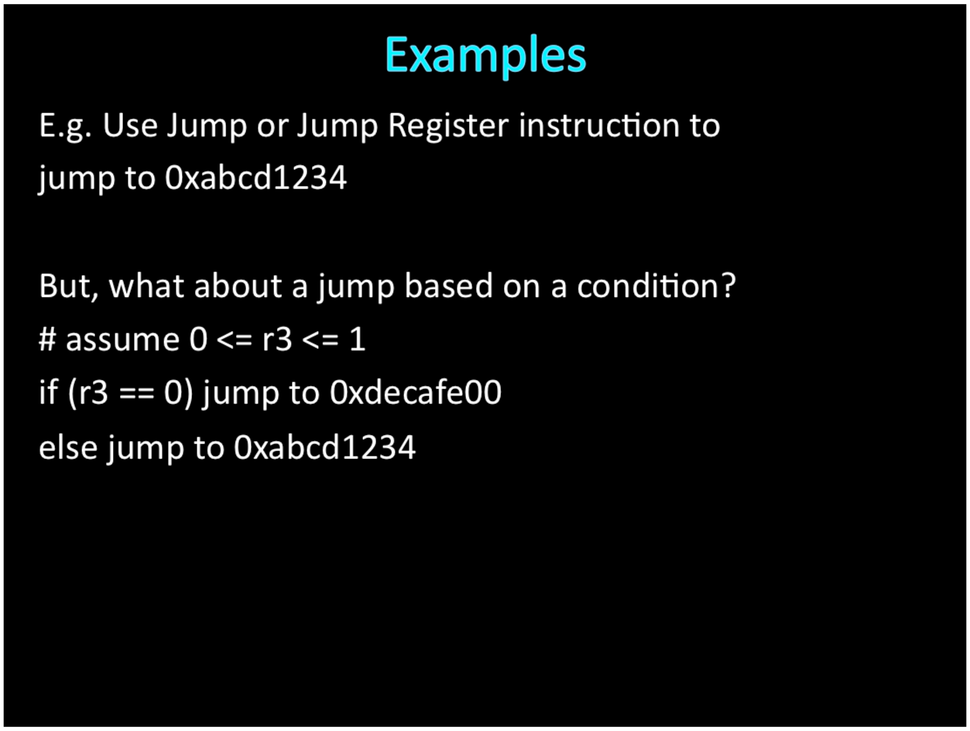
op	func	mnemonic	description
0x0	0x08	JR rs	PC = R[rs]

ex: JR r3

Jump Register



op	func	mnemonic	description
0x0	0x08	JR rs	PC = R[rs]

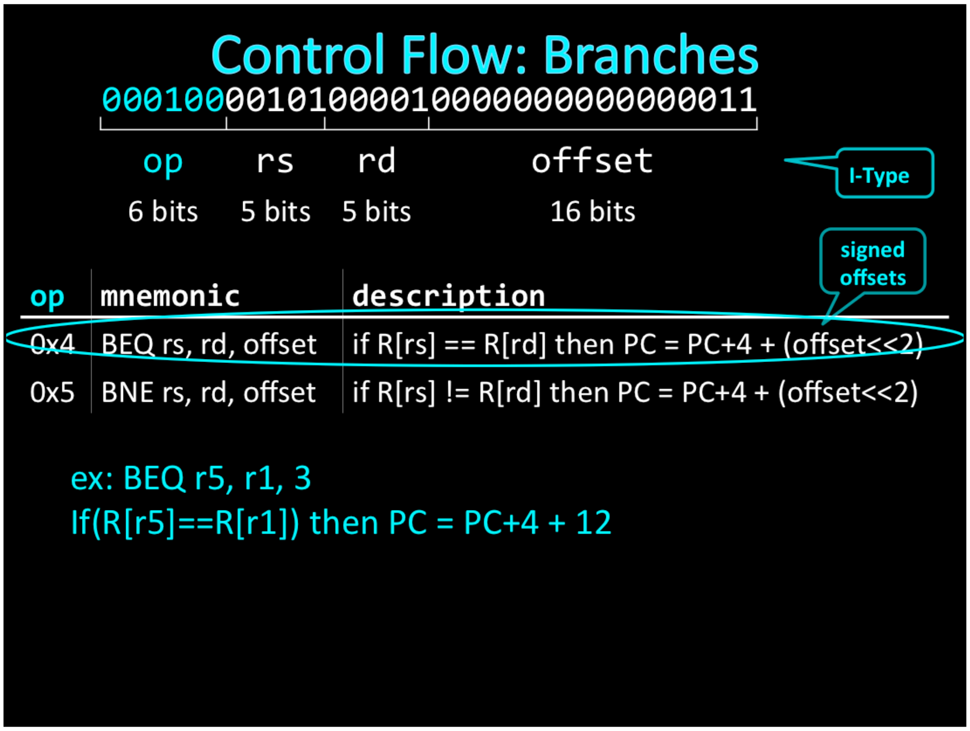


Examples

E.g. Use Jump or Jump Register instruction to jump to 0xabcd1234

But, what about a jump based on a condition?

```
# assume 0 <= r3 <= 1
if (r3 == 0) jump to 0xdecafe00
else jump to 0xabcd1234
```



Control Flow: Branches

```
00010000101000010000000000000011
```

op	rs	rd	offset
6 bits	5 bits	5 bits	16 bits

I-Type

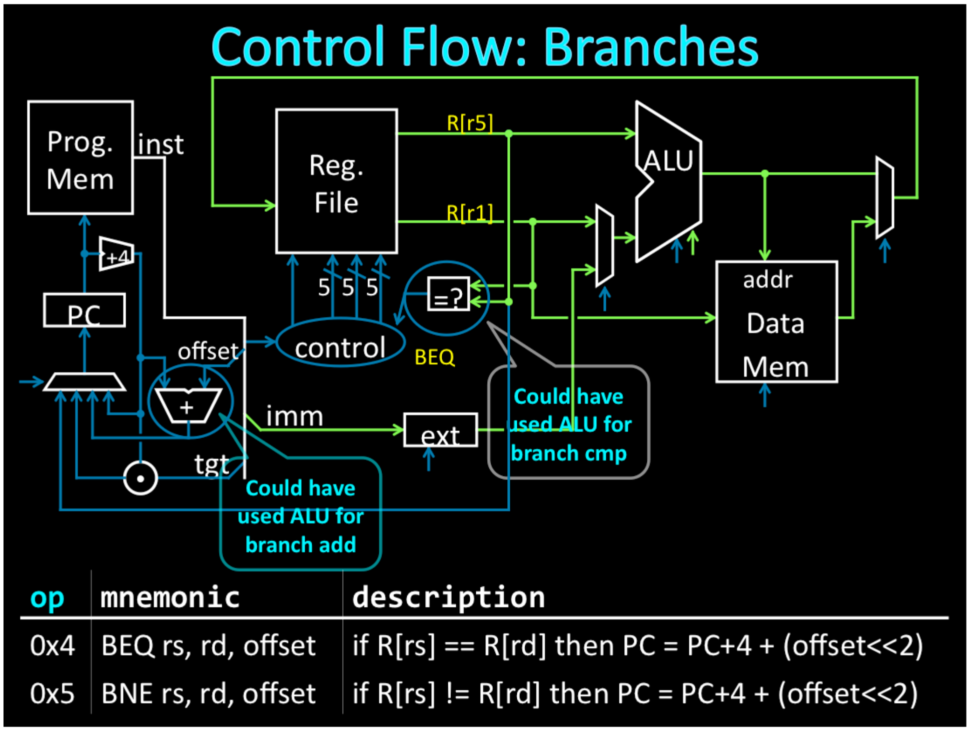
signed offsets

op	mnemonic	description
0x4	BEQ rs, rd, offset	if R[rs] == R[rd] then PC = PC+4 + (offset<<2)
0x5	BNE rs, rd, offset	if R[rs] != R[rd] then PC = PC+4 + (offset<<2)

ex: BEQ r5, r1, 3
If(R[r5]==R[r1]) then PC = PC+4 + 12

Why should the offset be left shifted by 2. To keep the jump address word aligned.

AND to not waste two bits that we *know* are going to be 0.



Control Flow: Branches

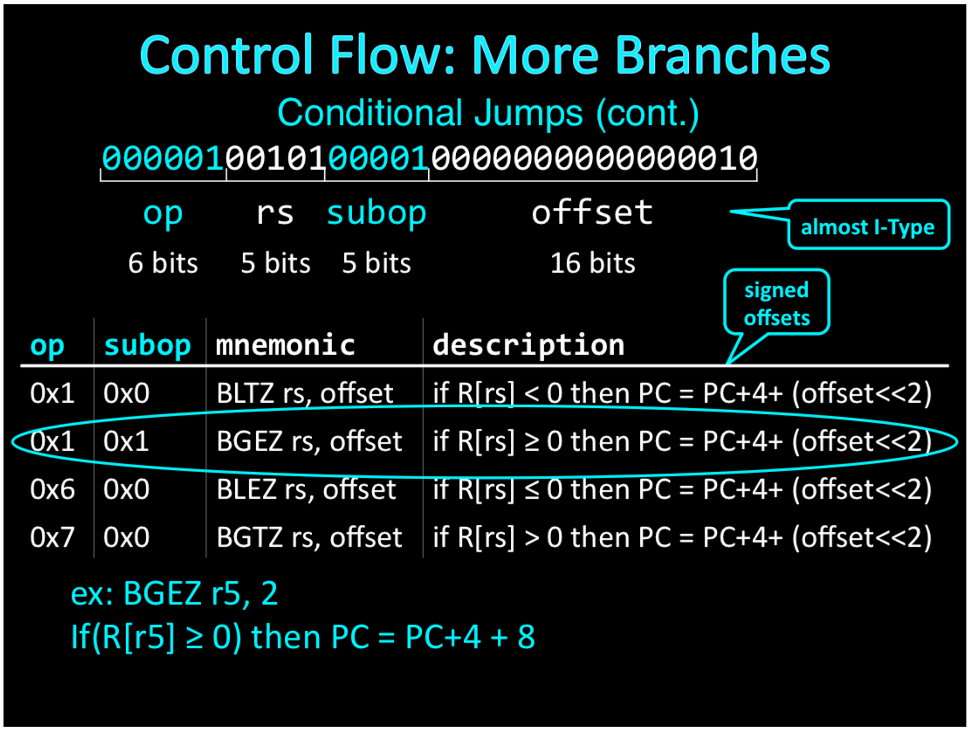
Could have used ALU for branch cmp

Could have used ALU for branch add

op	mnemonic	description
0x4	BEQ rs, rd, offset	if R[rs] == R[rd] then PC = PC+4 + (offset<<2)
0x5	BNE rs, rd, offset	if R[rs] != R[rd] then PC = PC+4 + (offset<<2)

In the book, the ALU is used to determine the branch comparison. We are choosing to do it separately.

Anyway, can’t use ALU for both branc comparison and PC determination. At most, we can use it once. The book uses it for branch



Control Flow: More Branches

Conditional Jumps (cont.)

$$\underbrace{000001}_{\substack{\text{op}\\\text{6 bits}}}\underbrace{00101}_{\substack{\text{rs}\\\text{5 bits}}}\underbrace{00001}_{\substack{\text{subop}\\\text{5 bits}}}\underbrace{0000000000000010}_{\substack{\text{offset}\\\text{16 bits}}}$$

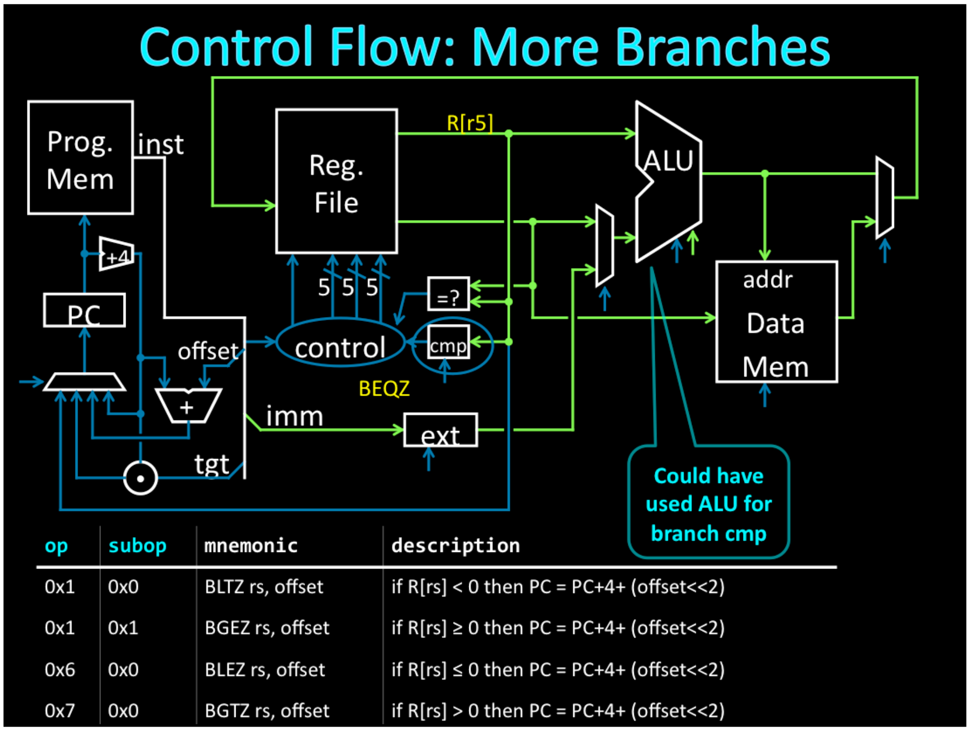
almost I-Type

signed offsets

op	subop	mnemonic	description
0x1	0x0	BLTZ rs, offset	if R[rs] < 0 then PC = PC+4+ (offset<<2)
0x1	0x1	BGEZ rs, offset	if R[rs] ≥ 0 then PC = PC+4+ (offset<<2)
0x6	0x0	BLEZ rs, offset	if R[rs] ≤ 0 then PC = PC+4+ (offset<<2)
0x7	0x0	BGTZ rs, offset	if R[rs] > 0 then PC = PC+4+ (offset<<2)

ex: BGEZ r5, 2

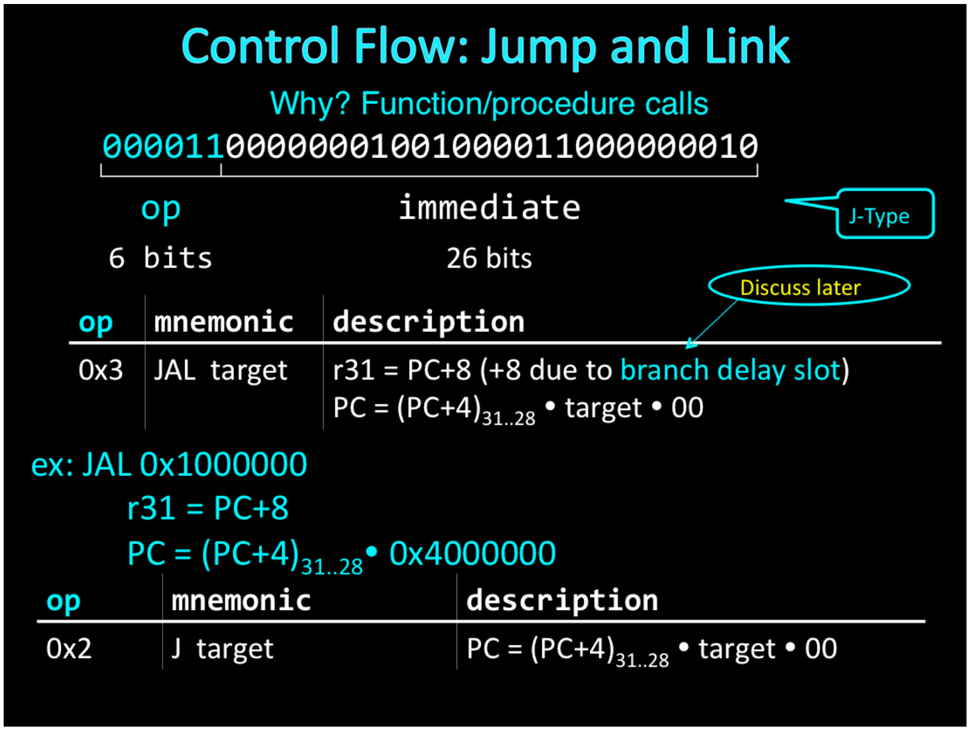
If(R[r5] ≥ 0) then PC = PC+4 + 8



Control Flow: More Branches

Could have used ALU for branch cmp

op	subop	mnemonic	description
0x1	0x0	BLTZ rs, offset	if R[rs] < 0 then PC = PC+4+ (offset<<2)
0x1	0x1	BGEZ rs, offset	if R[rs] ≥ 0 then PC = PC+4+ (offset<<2)
0x6	0x0	BLEZ rs, offset	if R[rs] ≤ 0 then PC = PC+4+ (offset<<2)
0x7	0x0	BGTZ rs, offset	if R[rs] > 0 then PC = PC+4+ (offset<<2)



Control Flow: Jump and Link

Why? Function/procedure calls

```
000011 00000001001000011000000010
|____||__________________________|
  op            immediate
6 bits           26 bits
```

J-Type

Discuss later

op	mnemonic	description
0x3	JAL target	r31 = PC+8 (+8 due to branch delay slot) PC = $(PC+4)_{31..28}$ • target • 00

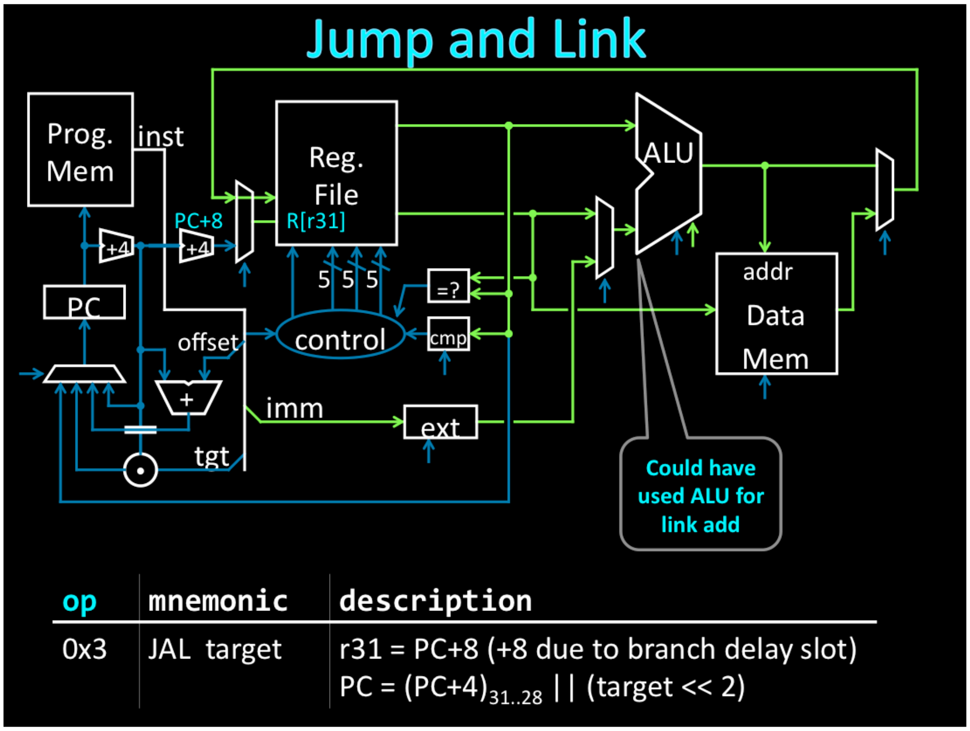
ex: JAL 0x1000000

r31 = PC+8

PC = $(PC+4)_{31..28}$• 0x4000000

op	mnemonic	description
0x2	J target	PC = $(PC+4)_{31..28}$ • target • 00

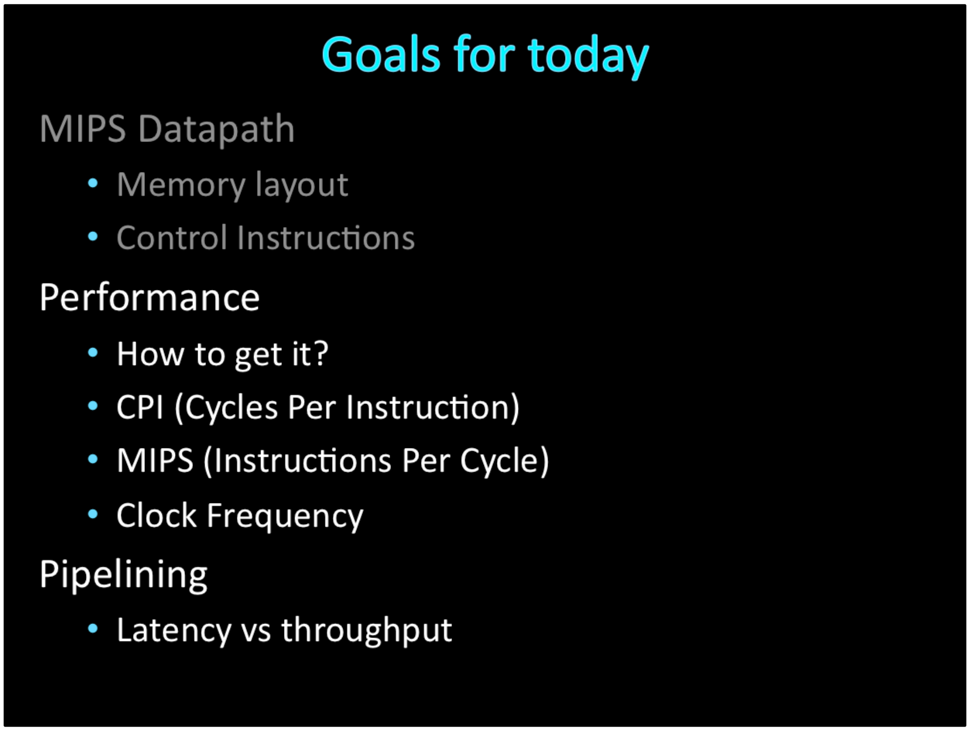
0x0121808 == 1185800



Jump and Link

Could have used ALU for link add

op	mnemonic	description
0x3	JAL target	r31 = PC+8 (+8 due to branch delay slot) PC = $(PC+4)_{31..28}$ || (target << 2)



Goals for today

MIPS Datapath

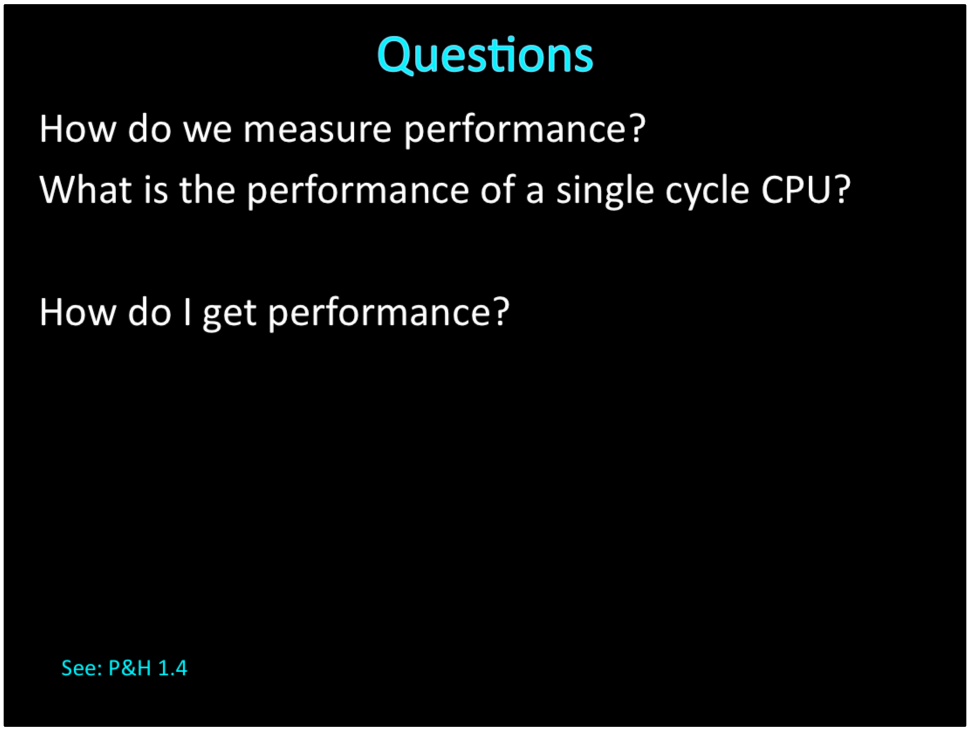
- Memory layout
- Control Instructions

Performance

- How to get it?
- CPI (Cycles Per Instruction)
- MIPS (Instructions Per Cycle)
- Clock Frequency

Pipelining

- Latency vs throughput



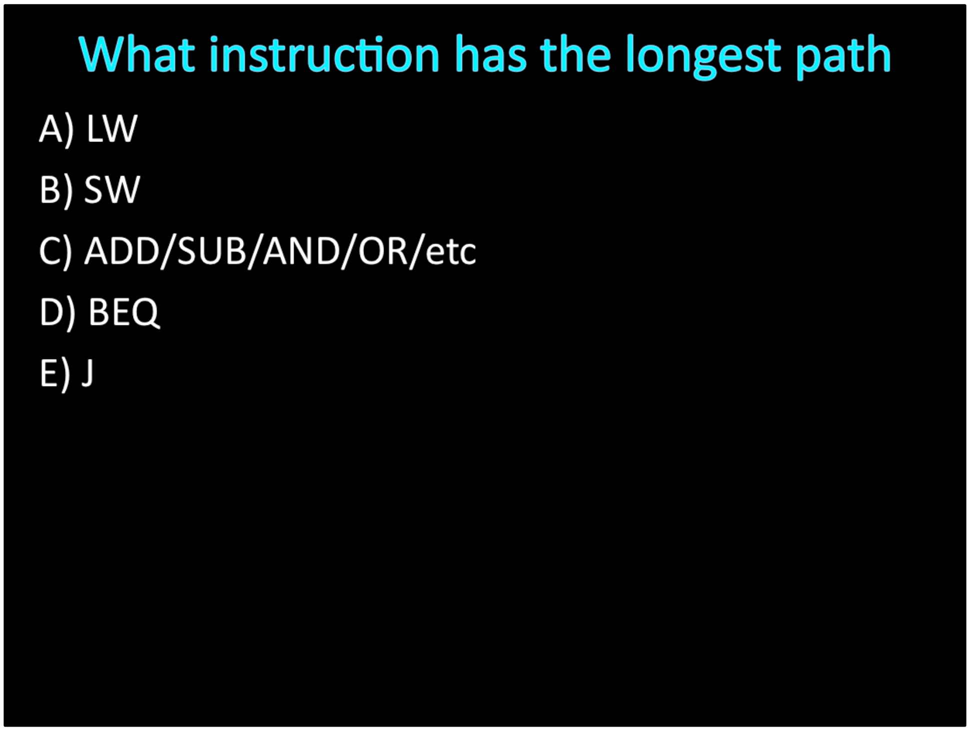
Questions

How do we measure performance?

What is the performance of a single cycle CPU?

How do I get performance?

See: P&H 1.4



What instruction has the longest path

A) LW

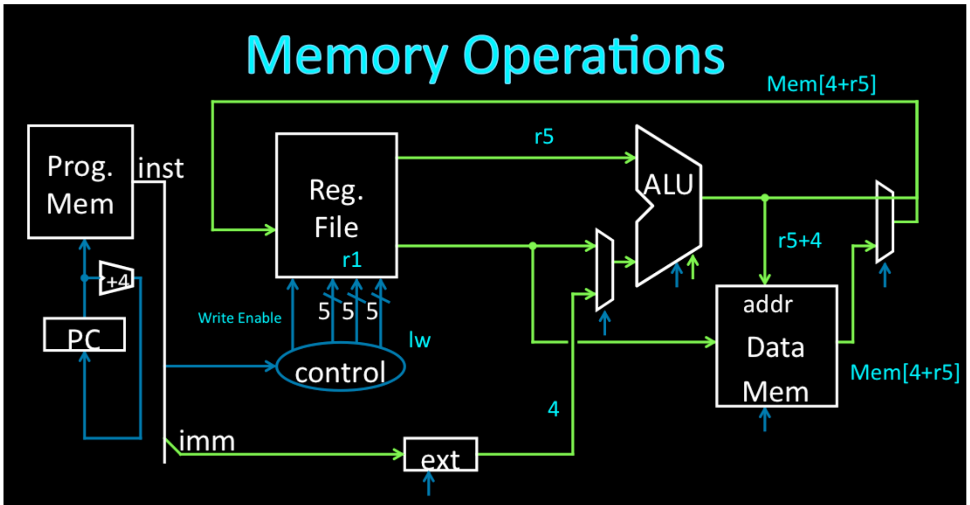
B) SW

C) ADD/SUB/AND/OR/etc

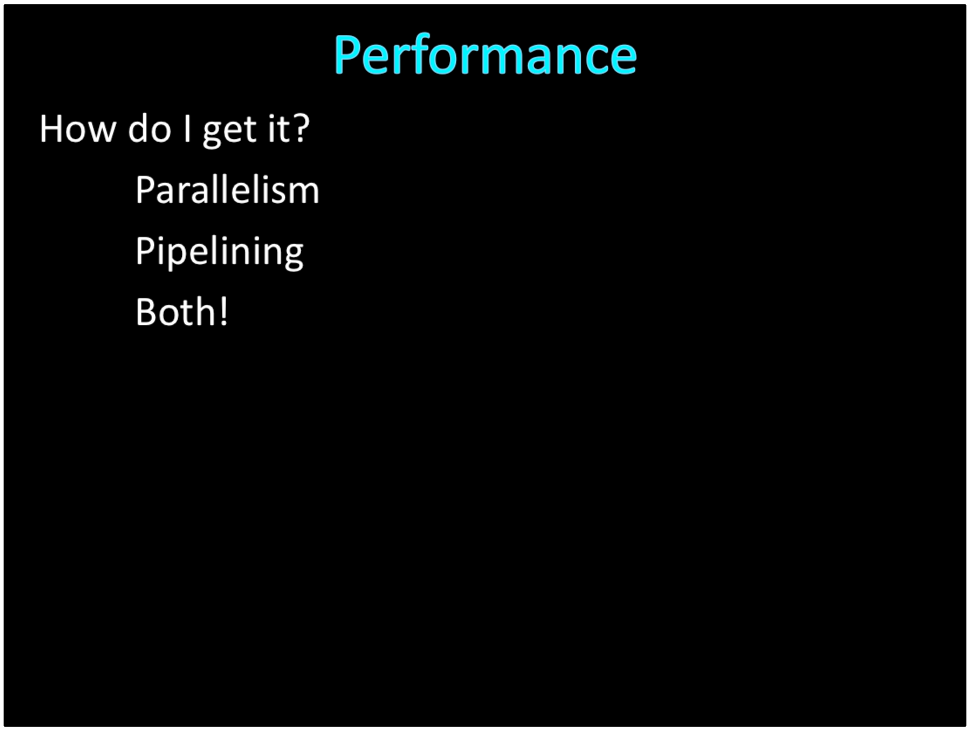
D) BEQ

E) J

Memory Operations



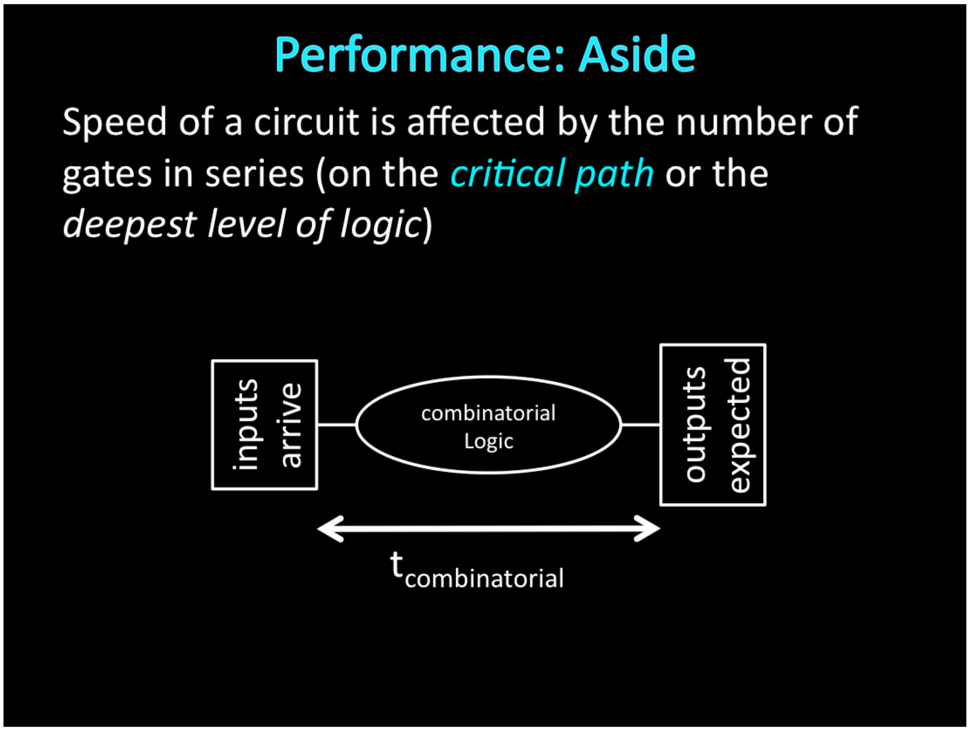
ex: = r1 = Mem[4+r5] # LW r1, 4(r5)



Performance

How do I get it?

- Parallelism
- Pipelining
- Both!



Performance: Aside

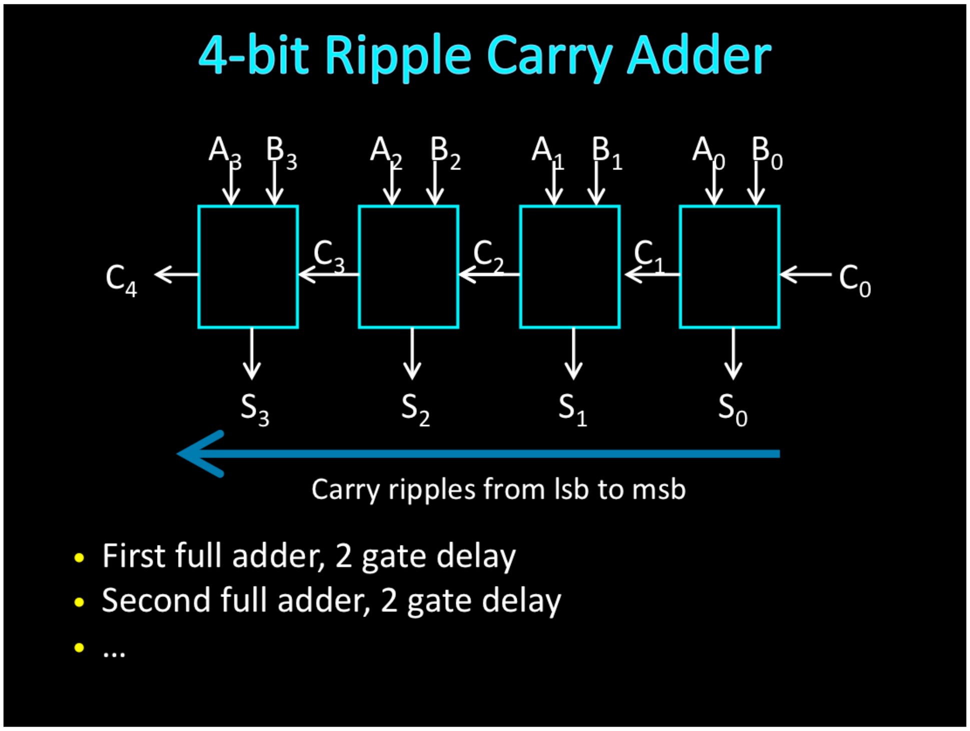
Speed of a circuit is affected by the number of gates in series (on the *critical path* or the *deepest level of logic*)

inputs arrive

combinatorial Logic

outputs expected

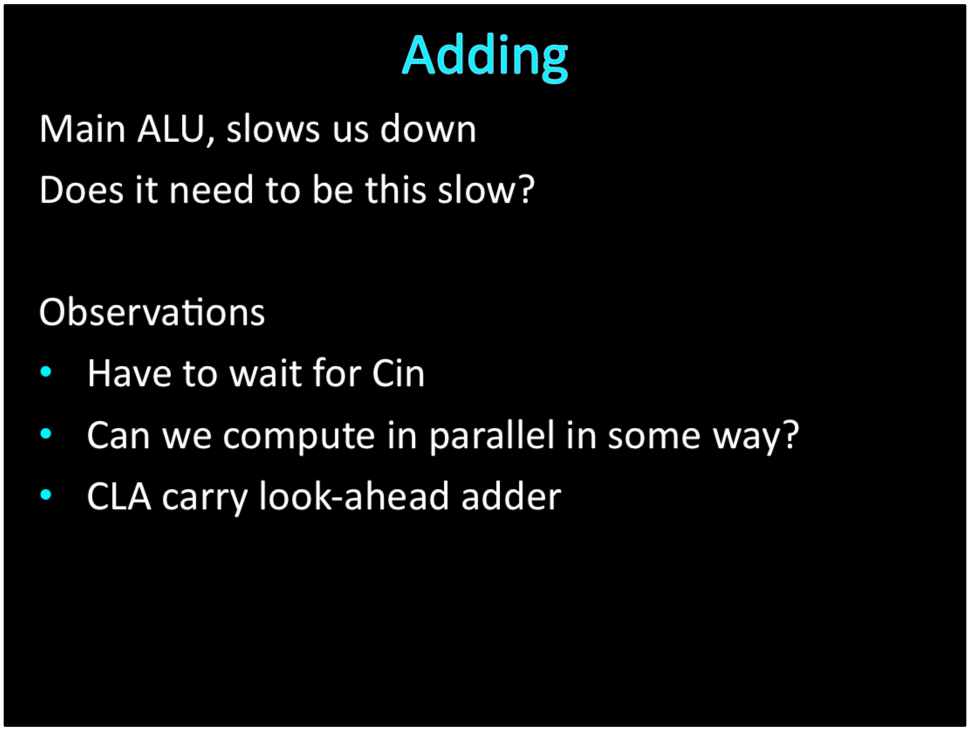
$t_{combinatorial}$



4-bit Ripple Carry Adder

Carry ripples from lsb to msb

- First full adder, 2 gate delay
- Second full adder, 2 gate delay
- …



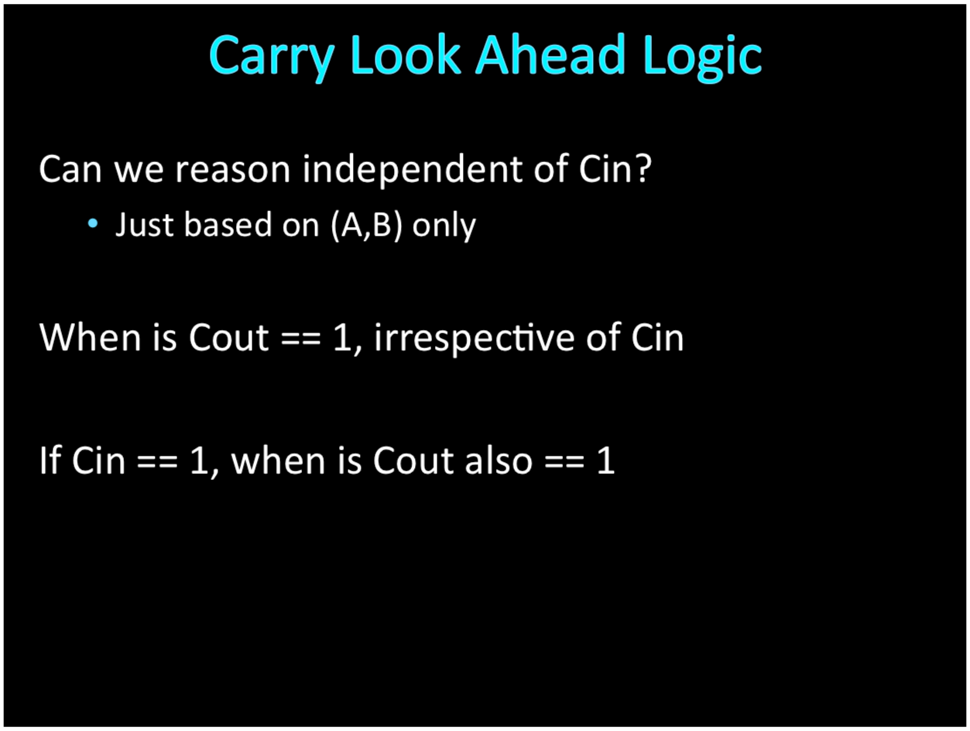
Adding

Main ALU, slows us down

Does it need to be this slow?

Observations

- Have to wait for Cin
- Can we compute in parallel in some way?
- CLA carry look-ahead adder



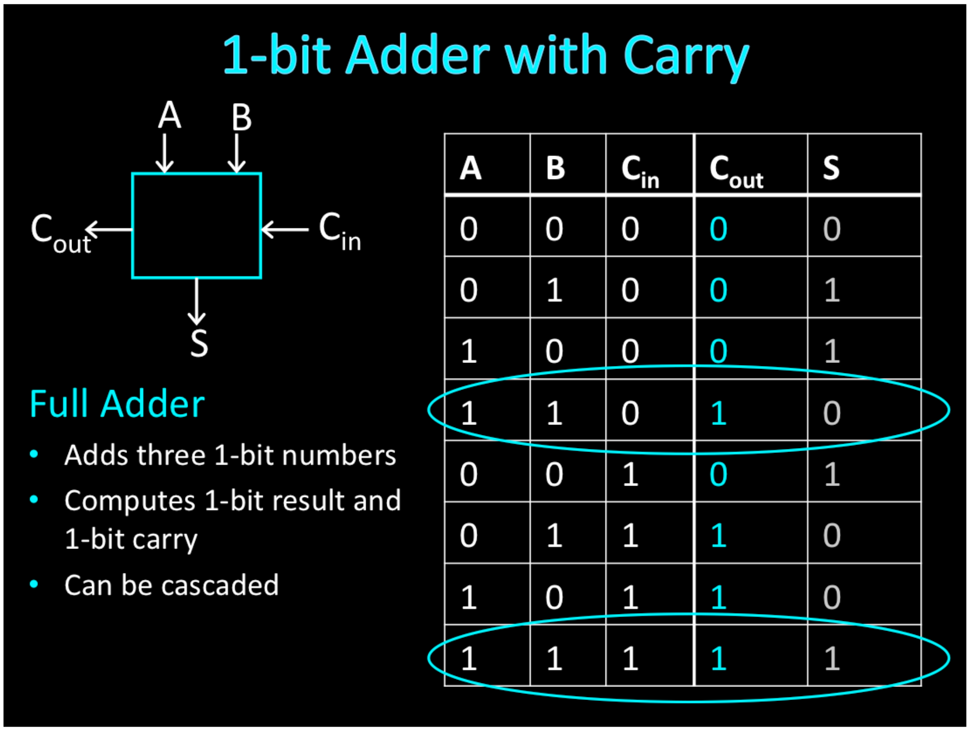
Carry Look Ahead Logic

Can we reason independent of Cin?

- Just based on (A,B) only

When is Cout == 1, irrespective of Cin

If Cin == 1, when is Cout also == 1



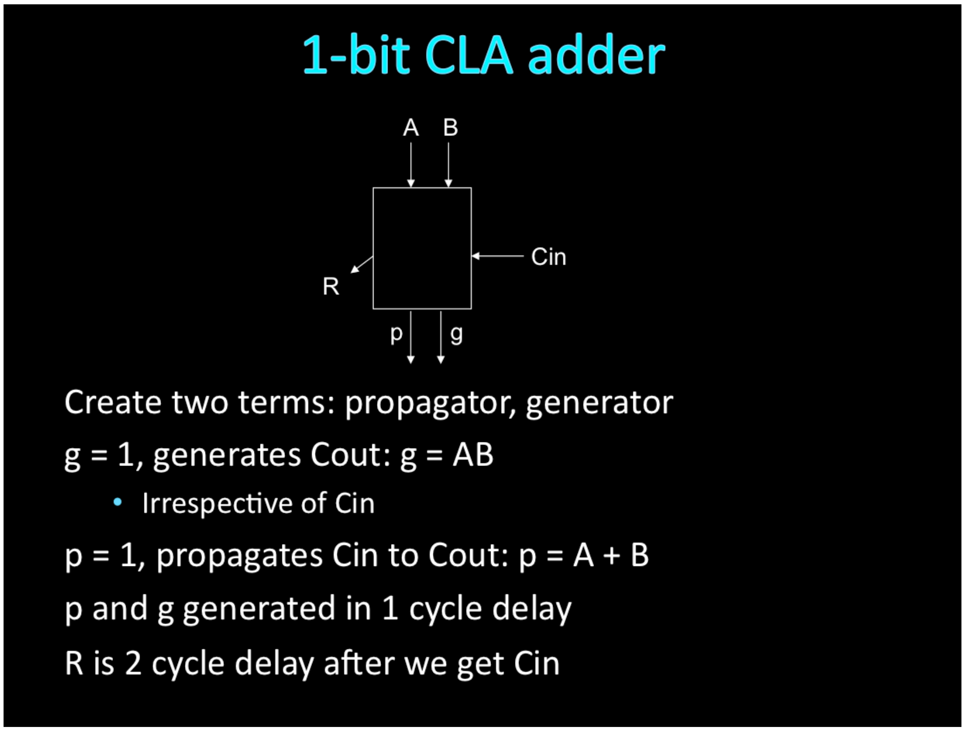
1-bit Adder with Carry

A
B
C$_{out}$
C$_{in}$
S

Full Adder

- Adds three 1-bit numbers
- Computes 1-bit result and 1-bit carry
- Can be cascaded

A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1



1-bit CLA adder

Create two terms: propagator, generator

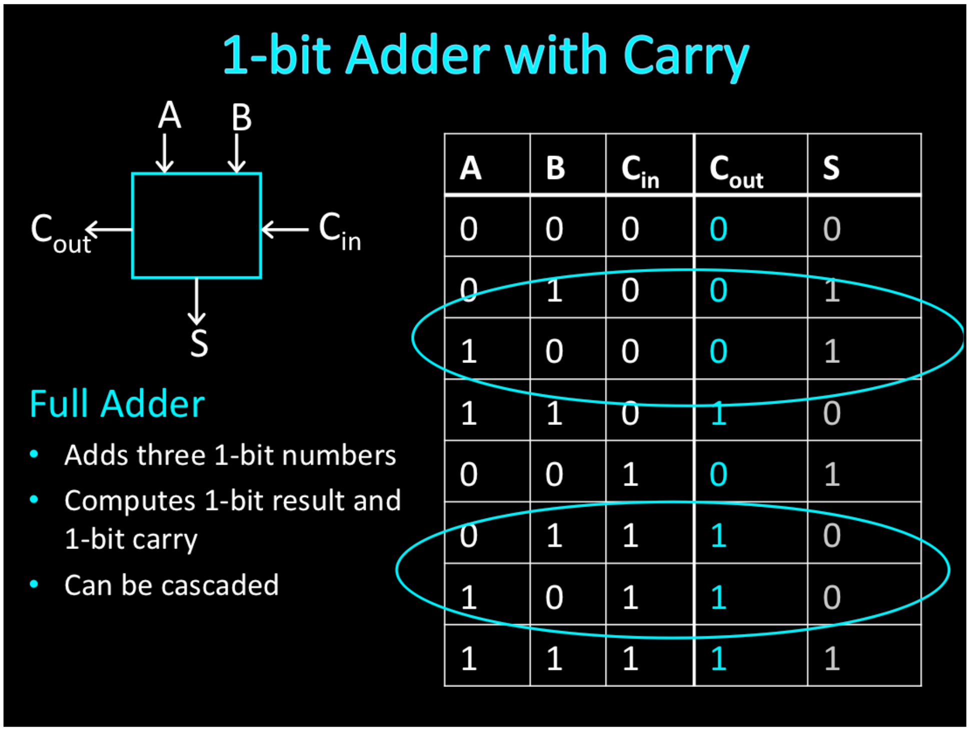
g = 1, generates Cout: g = AB

- Irrespective of Cin

p = 1, propagates Cin to Cout: p = A + B

p and g generated in 1 cycle delay

R is 2 cycle delay after we get Cin



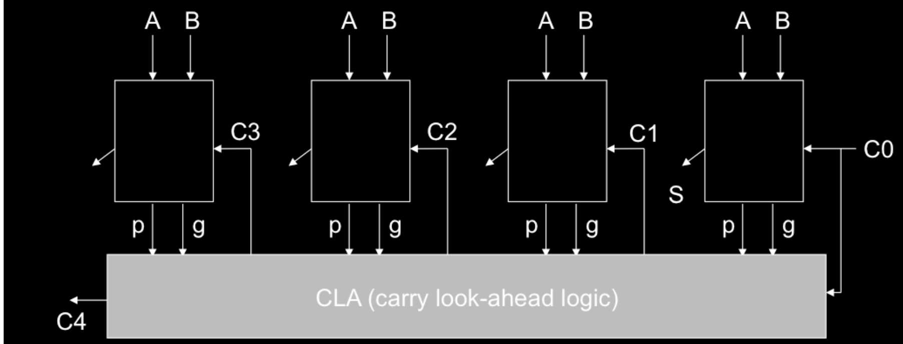
1-bit Adder with Carry

Full Adder

- Adds three 1-bit numbers
- Computes 1-bit result and 1-bit carry
- Can be cascaded

A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

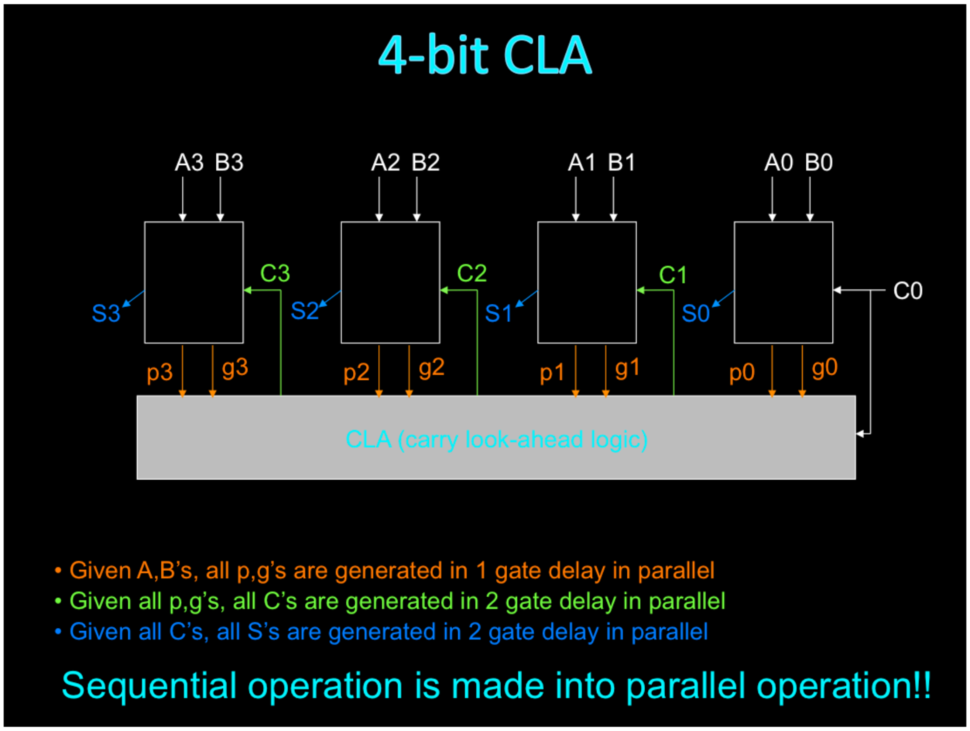
4-bit CLA



- CLA takes p,g from all 4 bits, C0 and generates all Cs: 2 gate delay

$$C_1 = g_0 + p_0C_0$$
$$C_2 = g_1 + p_1C_1 = g_1 + p_1(g_0 + p_0C_0) = g_1 + p_1g_0 + p_1p_0C_0$$
$$C_3 = g_2 + p_2C_2 = g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0C_0$$
$$C_4 = g_3 + p_3C_3 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0C_0$$

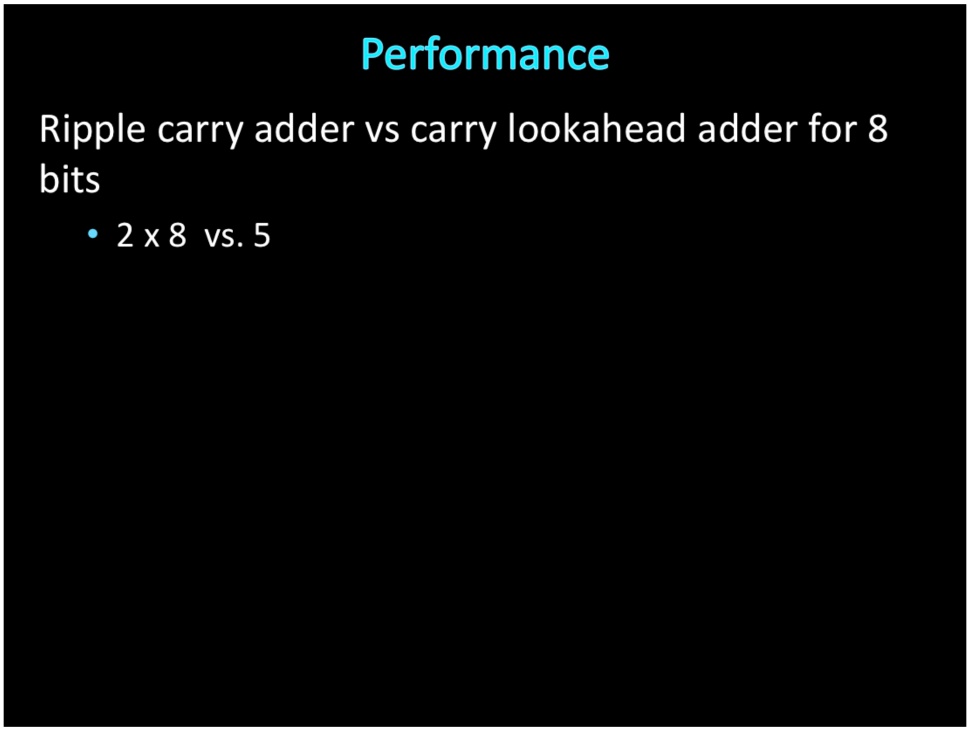
C_i = Function (C_0, p and g values)



4-bit CLA

- Given A,B's, all p,g's are generated in 1 gate delay in parallel
- Given all p,g's, all C's are generated in 2 gate delay in parallel
- Given all C's, all S's are generated in 2 gate delay in parallel

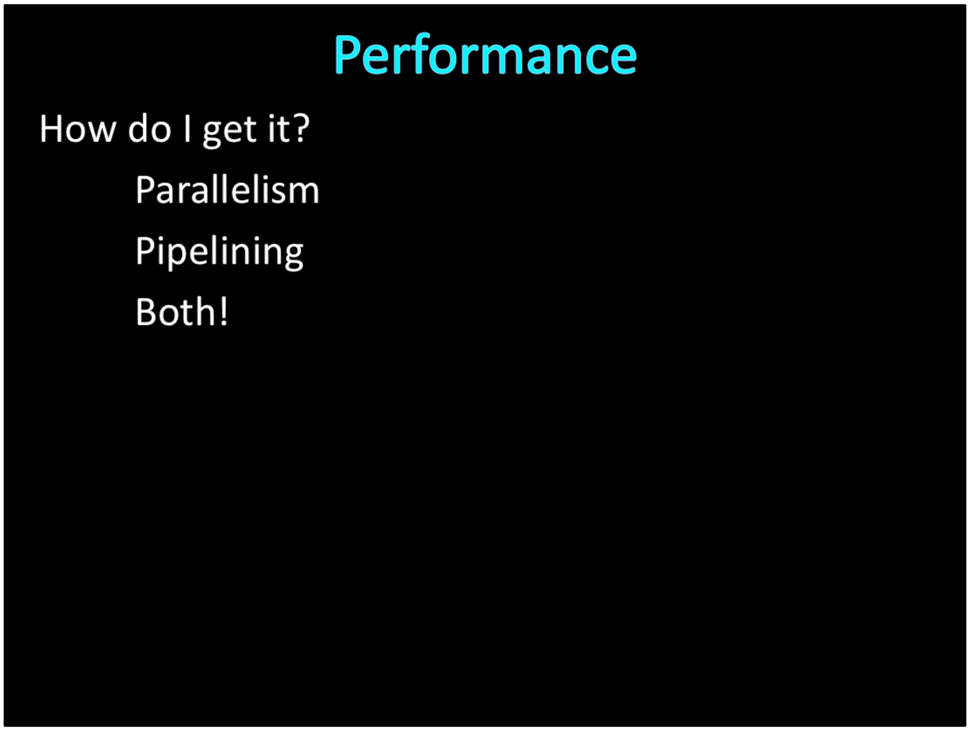
Sequential operation is made into parallel operation!!



Performance

Ripple carry adder vs carry lookahead adder for 8 bits

- 2 x 8 vs. 5



Performance

How do I get it?

- Parallelism
- Pipelining
- Both!