CS61C Midterm 2 Review Session

Problems + Solutions

Floating Point

Big Ideas in Number Rep

- Represent a large range of values by trading off precision.
- Store a number of fixed size (significand) and “float” its radix point around (exponent).

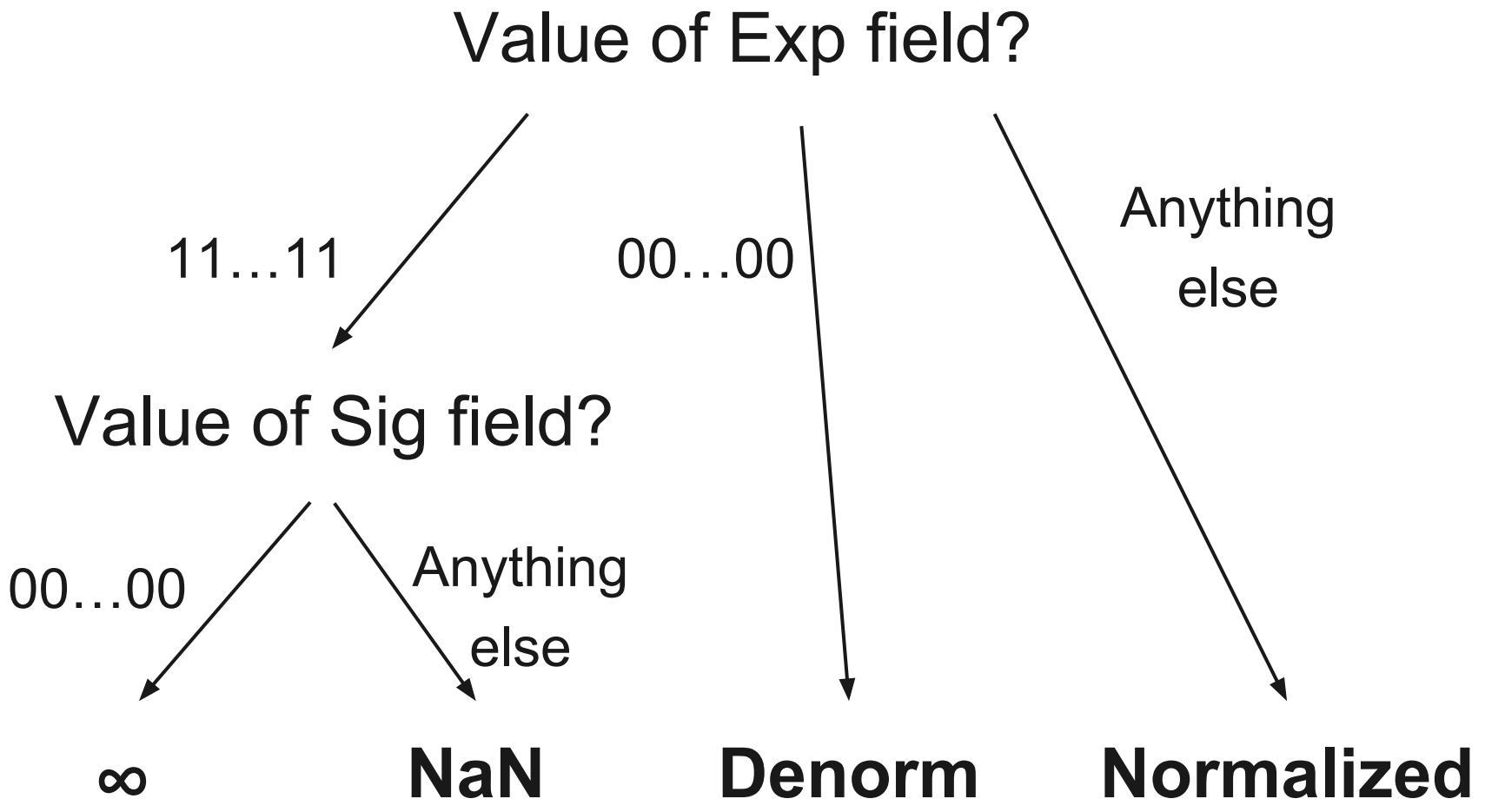
FP Format

$$= (-1)^{\text{Sign}} \times 2^{\text{Exp (from biased)}} \times (1 + \text{Sig})$$

for normalized numbers

- IEEE-754 32-bit float: 8 exp bits, 23 sig bits
- Same rules regardless of size!

Types of FP Numbers



Normalized vs. Denorm

Normalized number:

$$(-1)^{\text{Sign}} \times 2^{\text{Exp (from biased)}} \times (\mathbf{1} + \text{Sig})$$

Denormalized number:

$$(-1)^{\text{Sign}} \times 2^{\text{Exp (from biased)}\ \mathbf{+1}} \times (\mathbf{0} + \text{Sig})$$

Eg. For IEEE 754 float, 00000000_{biased} = -127,
so exponent is -126

FP Exercises 1

Format: [1-bit sign | 4-bit exp | 3-bit sig]

What is the bias?

Convert to FP:

2.5

Convert from FP:

0b01001011

FP Exercises 1

Format: [1-bit sign | 4-bit exp | 3-bit sig]

What is the bias? $2^{(4-1)} - 1 = 7$

Convert to FP:

2.5

Convert from FP:

0b01001011

FP Exercises 1

Format: [1-bit sign | 4-bit exp | 3-bit sig], bias=7

Convert to FP:

$2.5 = 2 + 0.5$

$= 2^{1} + 2^{-1}$

$= (1 + 2^{-2}) \times 2^{1}$

Sign: 0b0, Exp: 0b1000, Sig: 0b010

Result: 0b01000010

FP Exercises 1

Format: [1-bit sign | 4-bit exp | 3-bit sig], bias=7

Convert from FP:

0b01001011 $= 1 \times (1 + 2^{-2} + 2^{-3}) \times 2^{(9-7)}$

$= 2^2 + 2^0 + 2^{-1}$

$= 4 + 1 + 0.5$

$= 5.5$

FP Exercises 2

Format: [1-bit sign | 4-bit exp | 3-bit sig], bias=7

- What is the bit pattern of the smallest positive number representable?

- What is the smallest positive integer NOT representable?

FP Exercises 2

Format: [1-bit sign | 4-bit exp | 3-bit sig], bias=7

- What is the bit pattern of the smallest positive number representable?

Bit pattern: 0bXXXXXXXX

FP Exercises 2

Format: [1-bit sign | 4-bit exp | 3-bit sig], bias=7

- What is the bit pattern of the smallest **positive** number representable?

Bit pattern: 0b**0**XXXXXXX

FP Exercises 2

Format: [1-bit sign | 4-bit exp | 3-bit sig], bias=7

- What is the bit pattern of the smallest positive number representable?

Strategy: use the smallest **denorm**

Bit pattern: 0b0**0000**XXX

FP Exercises 2

Format: [1-bit sign | 4-bit exp | 3-bit sig], bias=7

- What is the bit pattern of the smallest positive number representable?

Strategy: use the **smallest** denorm

Final answer: 0b00000**001**

FP Exercises 2

Format: [1-bit sign | 4-bit exp | 3-bit sig], bias=7

- What is the smallest positive integer NOT representable?

Strategy: We know a float with significand bits $m_1 m_2 \ldots m_n$ (m_k = 0 or 1) equals:

$$(-1)^{\text{Sign}} \times (1 + 2^{-1} m_1 + 2^{-2} m_2 + \ldots + 2^{-n} m_n) \times 2^{\text{Exp}}$$

so if n > # of significand bits, we can't represent it

FP Exercises 2

Format: [1-bit sign | 4-bit exp | 3-bit sig], bias=7

- What is the smallest normalized positive integer NOT representable?

Thus we want $(-1)^{Sign} \times (1 + 2^{-n}) \times 2^{Exp}$, n = 4

Number is positive, so we want $(1 + 2^{-4}) \times 2^{Exp}$

FP Exercises 2

Format: [1-bit sign | 4-bit exp | 3-bit sig], bias=7

- What is the smallest normalized positive integer NOT representable?

$(1 + 2^{-4}) \times 2^{Exp} = 2^{Exp} + 2^{Exp - 4}$

Since, $2^{Exp - 4}$ must be an integer, and smallest value of $2^{Exp - 4}$ is $2^0 = 1$, Exp = 4

Thus, $2^{Exp} + 2^{Exp - 4} = 2^4 + 2^0 = 17$

Digital Logic

Boolean Algebra

OR is + (e.g. A+B)

AND is · or simply juxtaposed (e.g. A·B, AB)

NOT is ~ or an overline (e.g. ~A, $\overline{A}$)

OR and AND are commutative and associative.

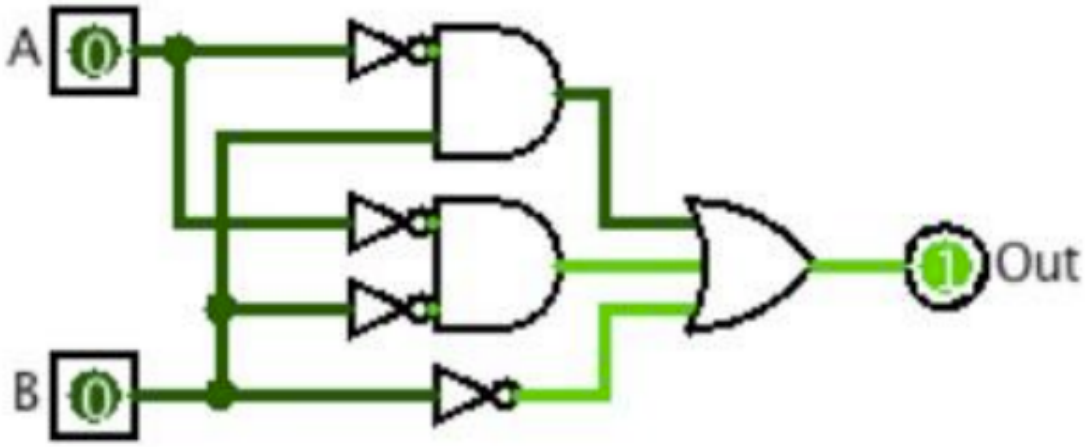
De Morgan's Laws

$\overline{A \cdot B} = \overline{A} + \overline{B}$

$\overline{A + B} = \overline{A} \cdot \overline{B}$

Warm-Up: Circuit Simplification

Rebuild this circuit with the fewest gates, using **only** AND, OR and NOT gates.

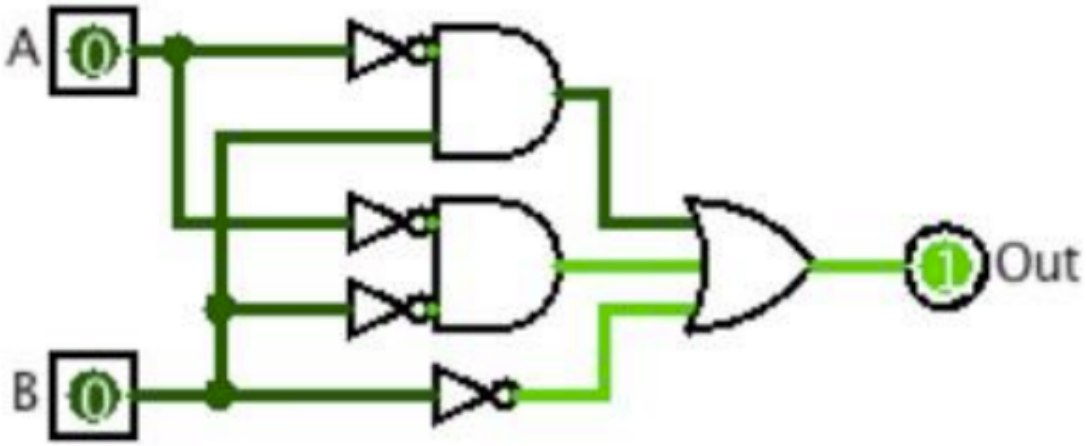


Warm-Up: Circuit Simplification

~A * B + ~A * ~B + ~B

= ~A (B + ~B) + ~B

= ~A + ~B



Delays

- **Setup Time**: time needed for the input to be stable before the rising edge of the clock.
- **Hold Time**: time needed for the input to be stable after the rising edge of the clock.
- **CLK-to-Q Delay**: the time needed for the input of the register to be passed to the output of the register after the rising edge of the clock.

Delays

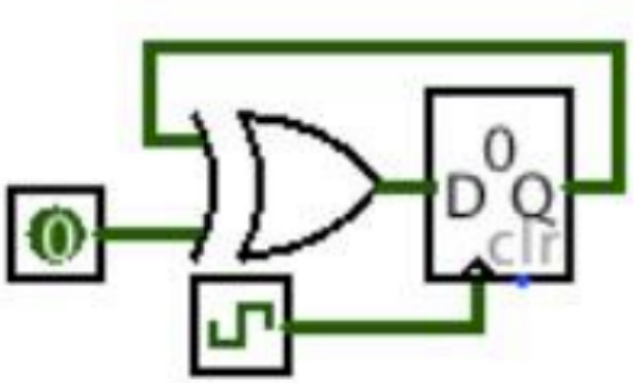
- Max Delay: Delay between two registers.
  - Also known as the **critical path time**.
- Max Delay = CLK-to-Q Delay
  + Combinational Logic Delay
  + Setup Time
- Max Frequency = 1/Max Delay

Warm-Up: Clocking

Choose an XOR gate for the circuit below. The clock speed is 2GHz (1/(500ps)); the setup, hold, and clock-to-q times of the register are 40, 70, and 60 picoseconds (10^{-12} s) respectively. Assume the input comes from a clocked register as well.

What range of XOR gate delays is acceptable?

e.g., “at least W ps”,
“at most X ps”, or “Y to Z ps”.



Warm-Up: Clocking

Max Delay = 500ps

Max Delay = CLK to Q + XOR Delay + Setup Time

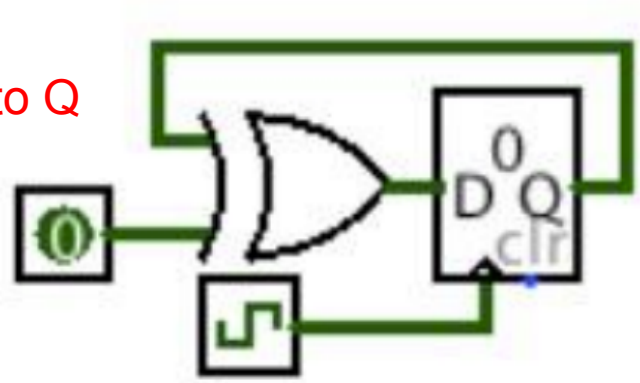
XOR Delay = 500 - 40 - 60

XOR Delay = 400 ps

What if XOR Delay = 0 ps?

You'll violate the Hold Time!

Need an XOR delay of at least Hold Time - CLK to Q = 10 ps



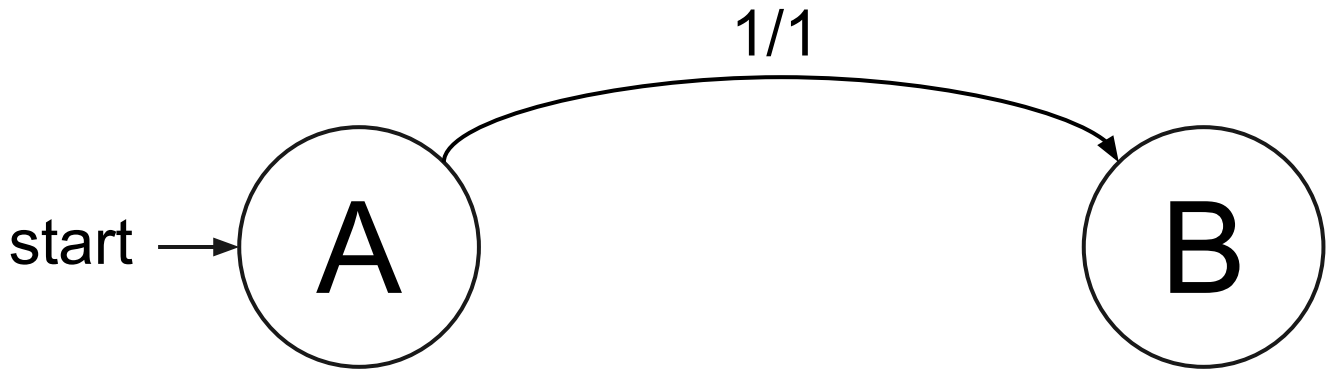
10 <= XOR Delay <= 400 ps

Using as few states as possible, complete the following finite state machine that takes a ternary (base-3) digit as input (0, 1, or 2). This machine should output a 1 if the sequence of ternary digits forms an odd number, otherwise it should output a 0.

Example: 1, 1, 2 $\rightarrow$ 112_3 (14_{10}) $\rightarrow$ even

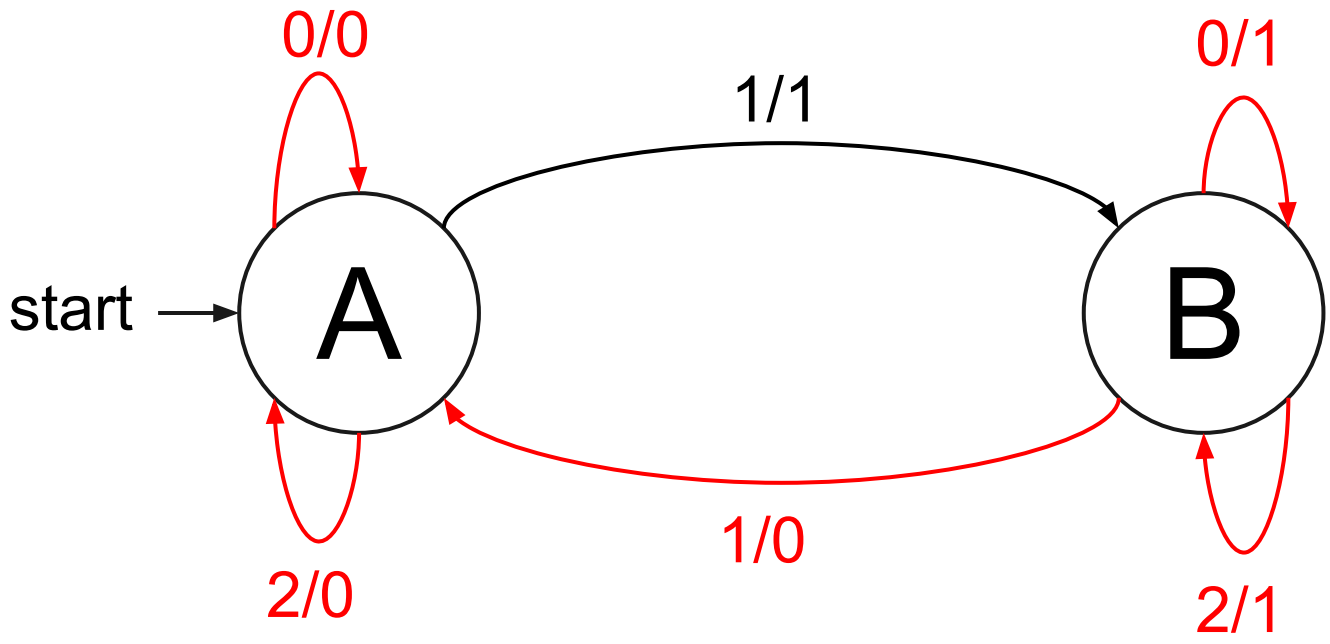
Past Exam Question (Garcia Fall 2014)

Input: ternary digit
Output: 1 if sequence is odd, 0 if even.



Past Exam Question (Garcia Fall 2014)

Input: ternary digit
Output: 1 if sequence is odd, 0 if even.



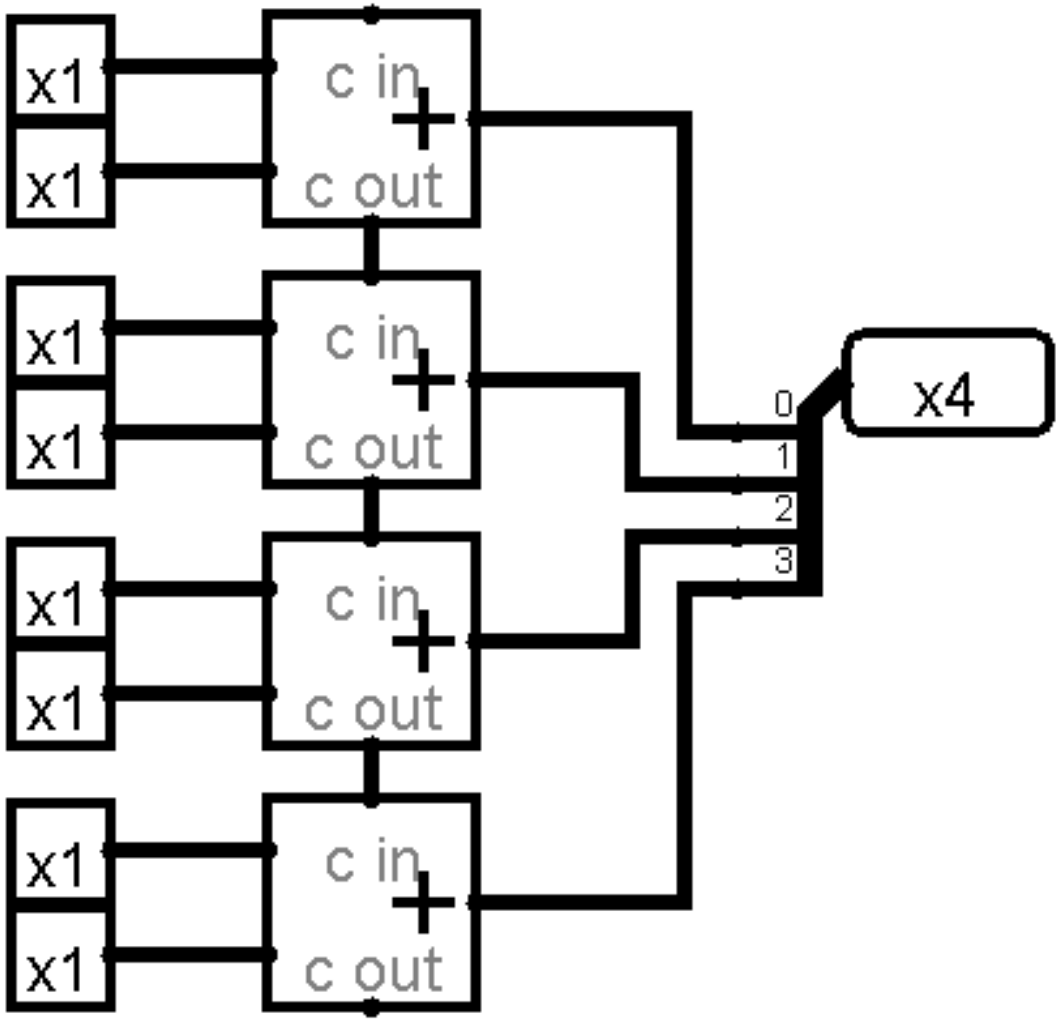
Past Exam Question (Garcia Fall 2014)

If the delay through a single-bit adder is:

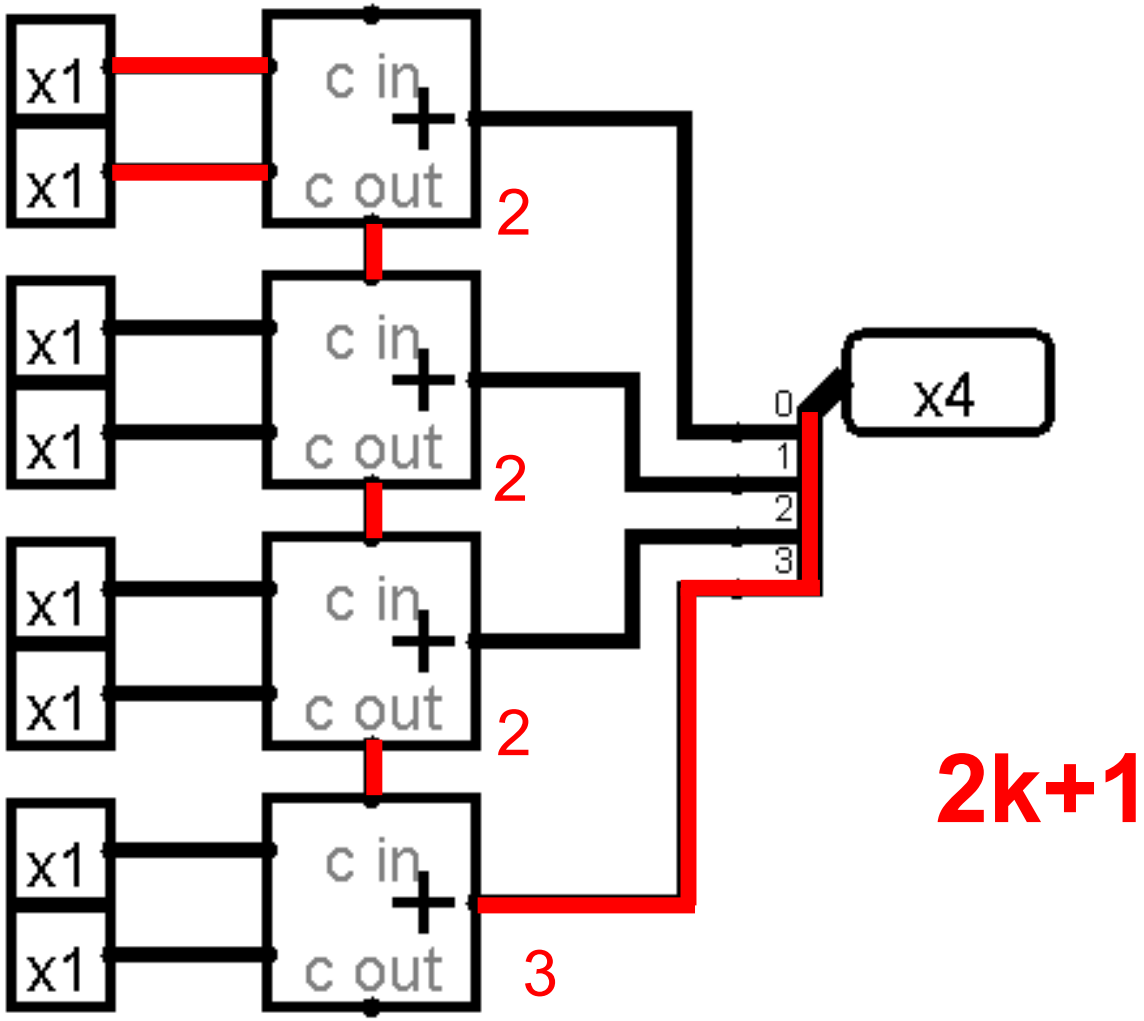
- 3 (measured in gate delays) to the sum output
- 2 to the carry output

What is the delay through a k-bit ripple-carry adder?

Past Exam Question (Garcia Fall 2014)



Past Exam Question (Garcia Fall 2014)

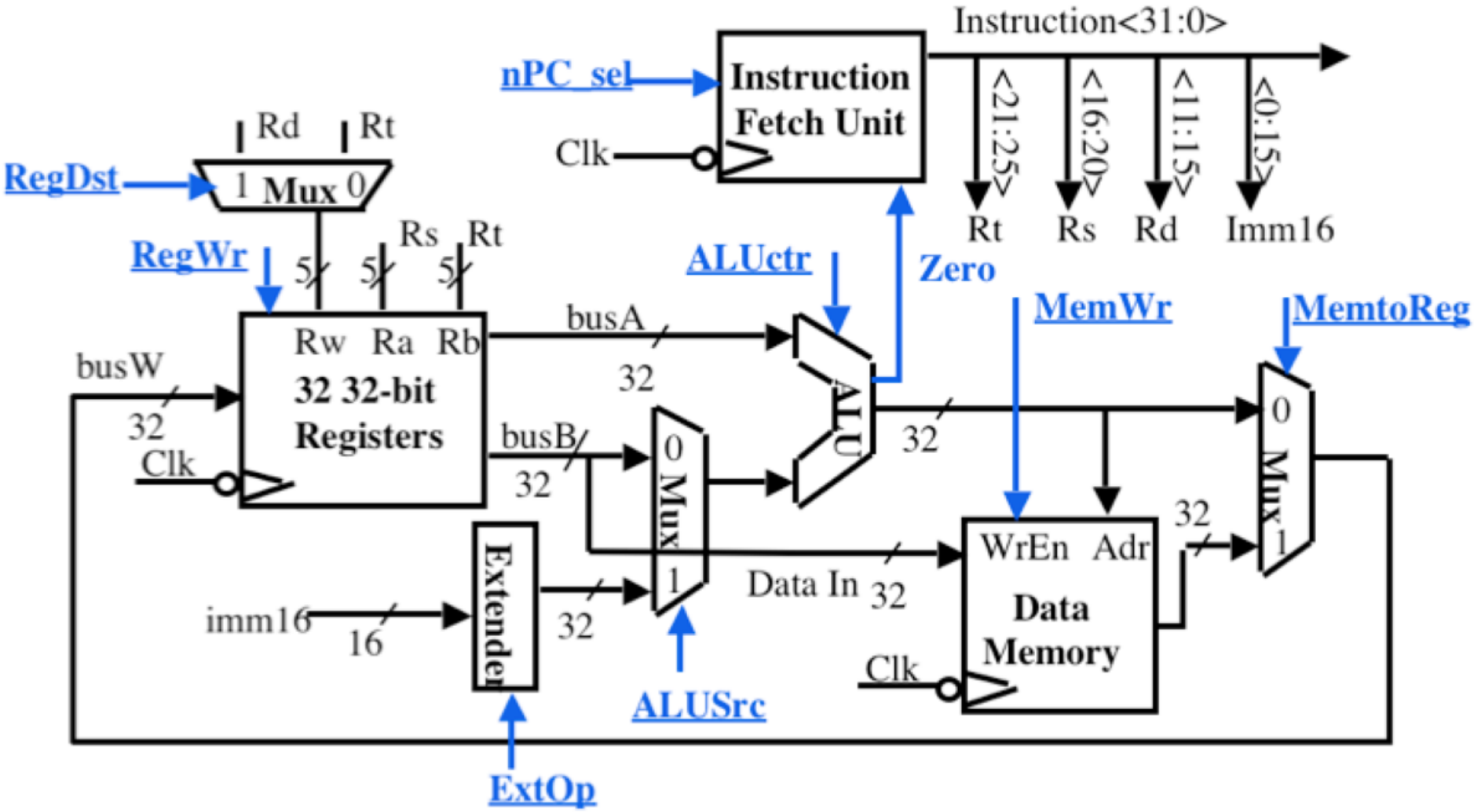


Past Exam Question (Garcia Fall 2014)

CPU Architecture

Datapath & Control

- Be able to trace the execution of MIPS instructions through the CPU
- Understand how the new PC value is calculated
- Know how to modify datapath & control signals to implement a new instruction


Instruction<31:0>
nPC_sel
Instruction Fetch Unit
Clk
<21:25>
<16:20>
<11:15>
<0:15>
Rt
Rs
Rd
Imm16
Rd
Rt
RegDst
1 Mux 0
RegWr
5
5
5
Rs
Rt
ALUctr
Zero
MemWr
MemtoReg
busA
32
Rw Ra Rb
32 32-bit Registers
busW
32
Clk
busB
32
0 Mux 1
ALU
32
0 Mux 1
32
WrEn Adr
Data In
32
Data Memory
Clk
imm16
16
Extender
32
ALUSrc
ExtOp

Example: Triple Add

New instruction: `add3 $rd,$rs,$rt`

Adds R[rs], R[rt], R[rd] and stores it into R[rd]

- Which MIPS instruction type would be best to represent `add3`?

- What is the register transfer level (RTL)?

Example: Triple Add

Adds R[rs], R[rt], R[rd] and stores it into R[rd]

- Which MIPS instruction type would be best to represent `add3`?

  R-type

- What is the register transfer level (RTL)?

  R[rd] = R[rs] + R[rt] + R[rd]; PC = PC + 4

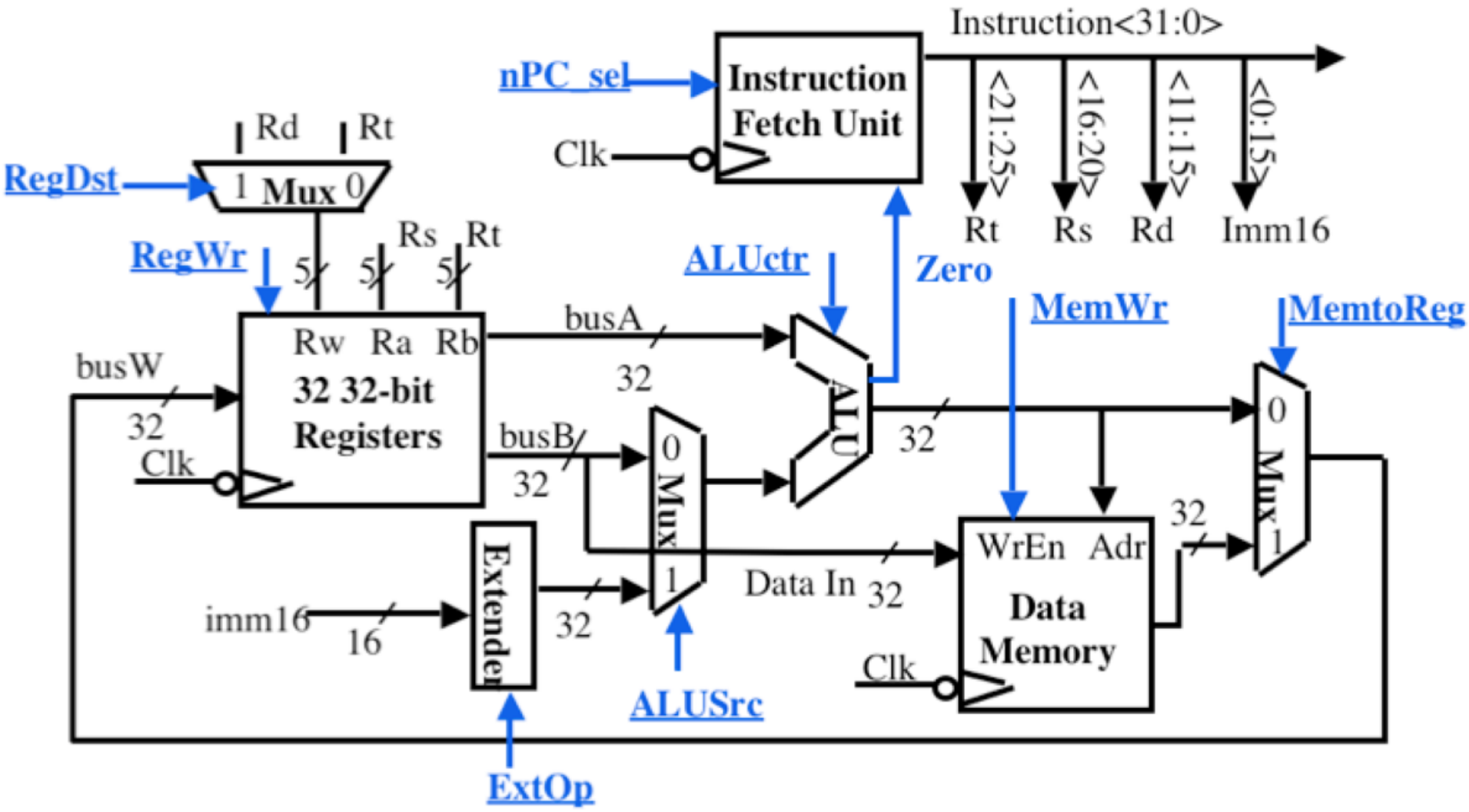
Triple Add: Datapath

Adds R[rs], R[rt], R[rd] and stores it into R[rd]

Make the **minimal** amount of changes on the datapath (next slide). Assume that the regfile has an additional read port for R[rd].

You may only add wires, muxes, and adders.

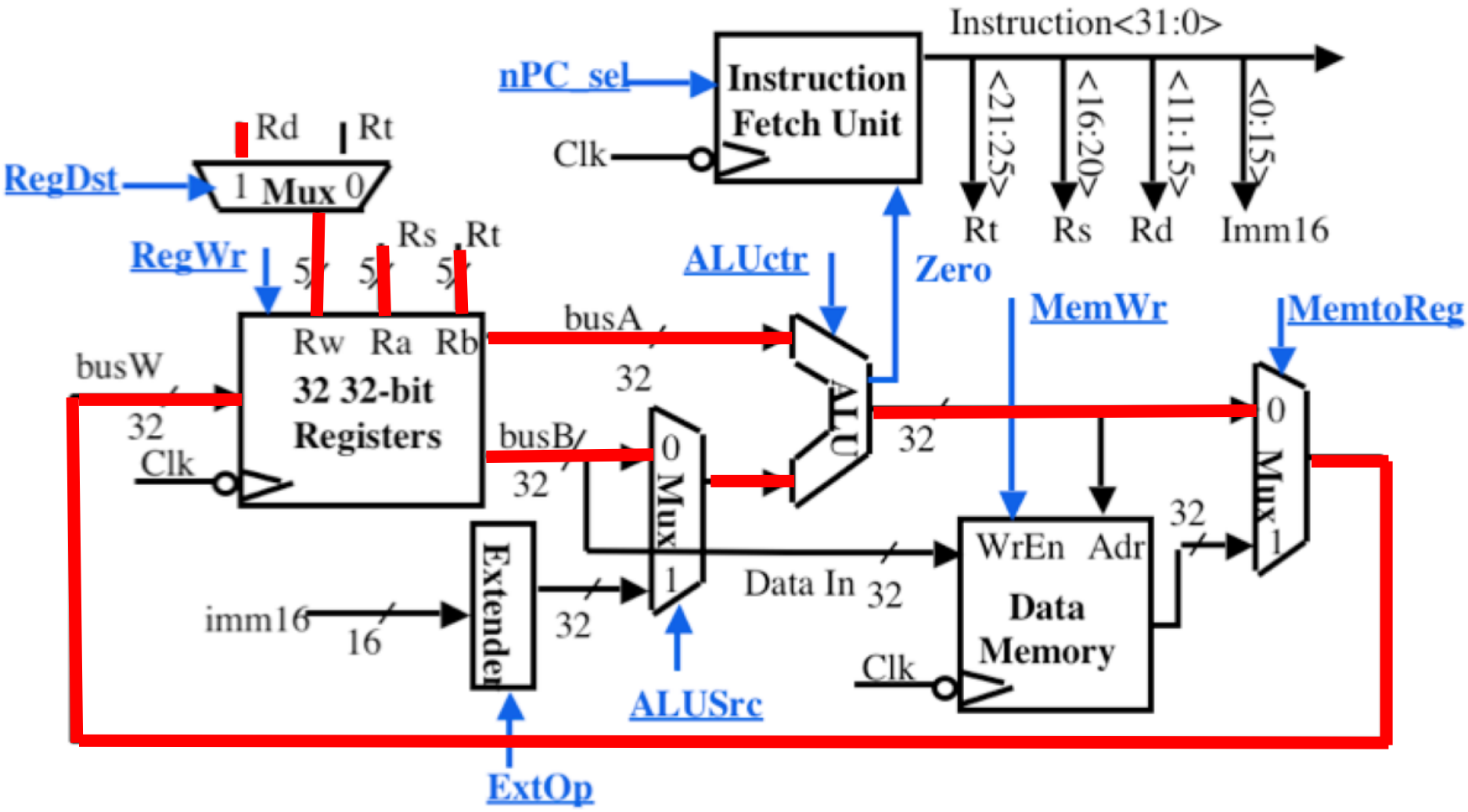
R[rd] = R[rs] + R[rt] + R[rd]; PC = PC + 4



Triple Add: Datapath

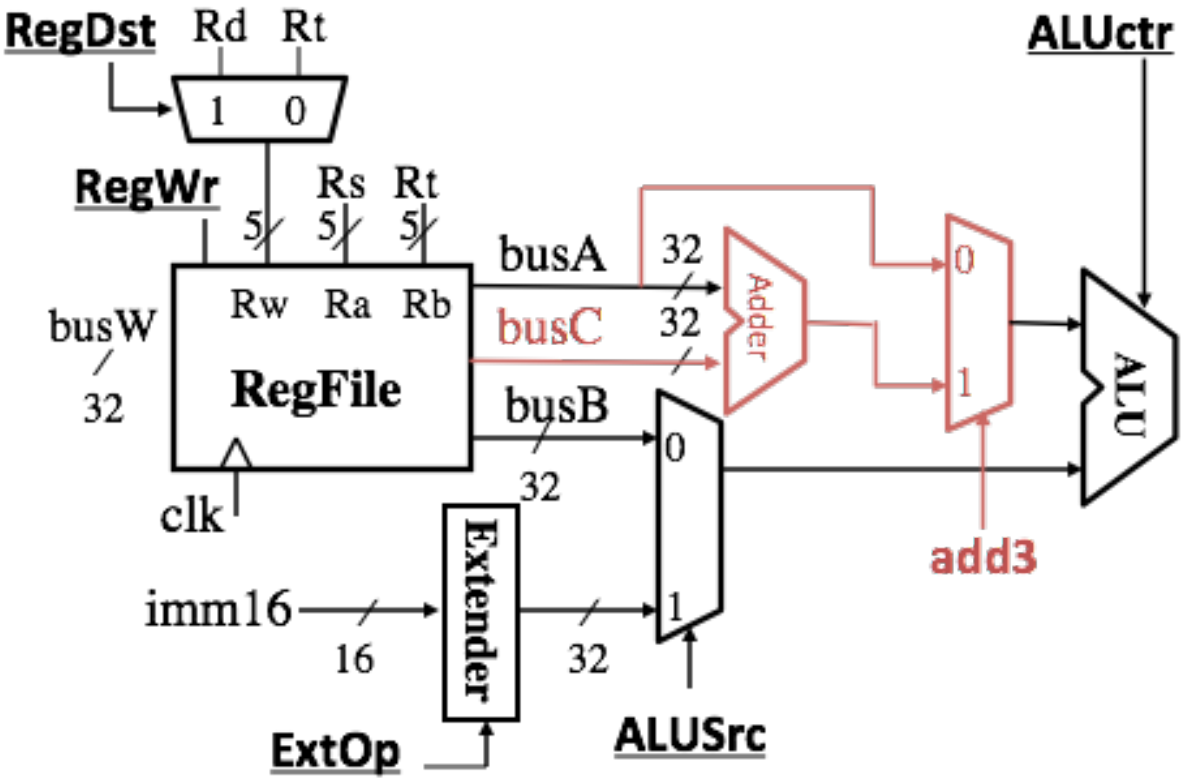
1. Identify existing components that helps implement the instruction.
2. Of the components NOT used, can any be used in the instruction?
3. Create components for anything that's not yet implemented.
4. Wire everything together, adding muxes/control signals as needed.

R[rd] = R[rs] + R[rt] + R[rd]; PC = PC + 4

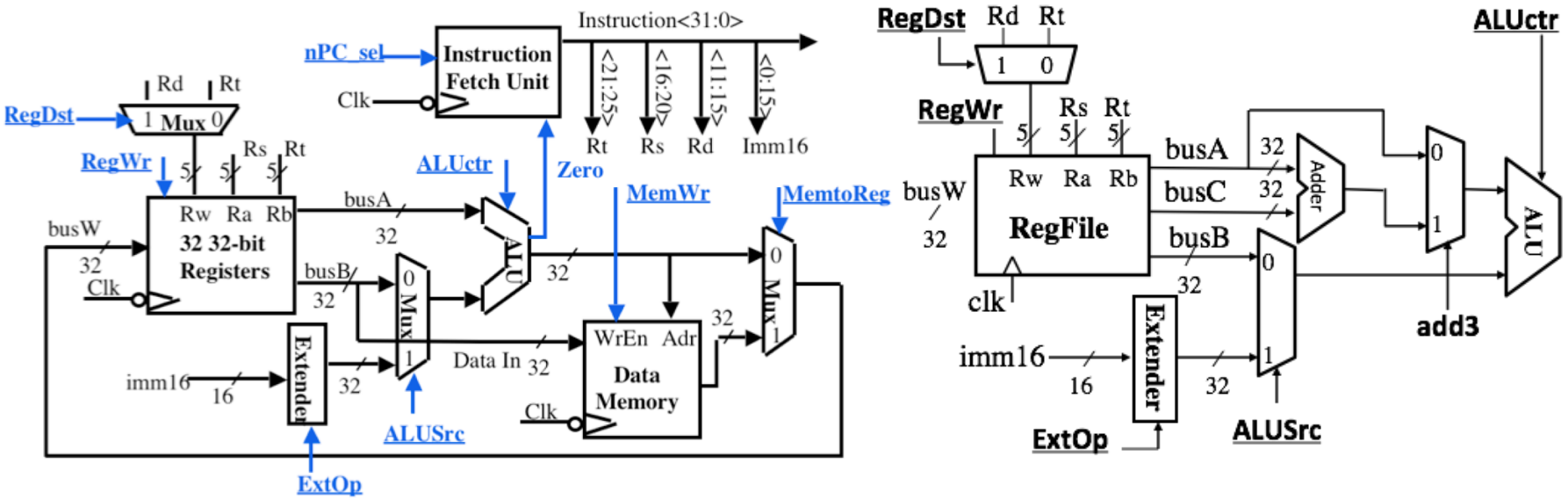


Triple Add: Datapath

Added Components:



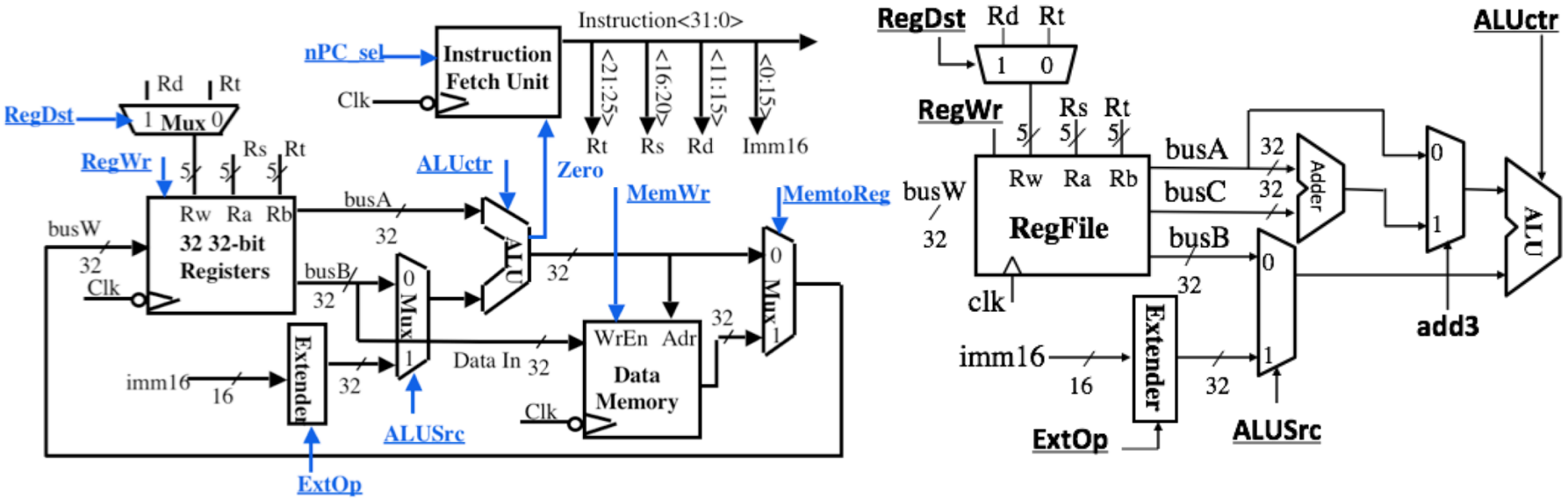
Triple Add: Control



Fill in the control signals:

RegDst	RegWr	nPC_sel	ExtOp	ALUSrc	ALUCtr	MemWr	Mem2Reg	add3

Triple Add: Control

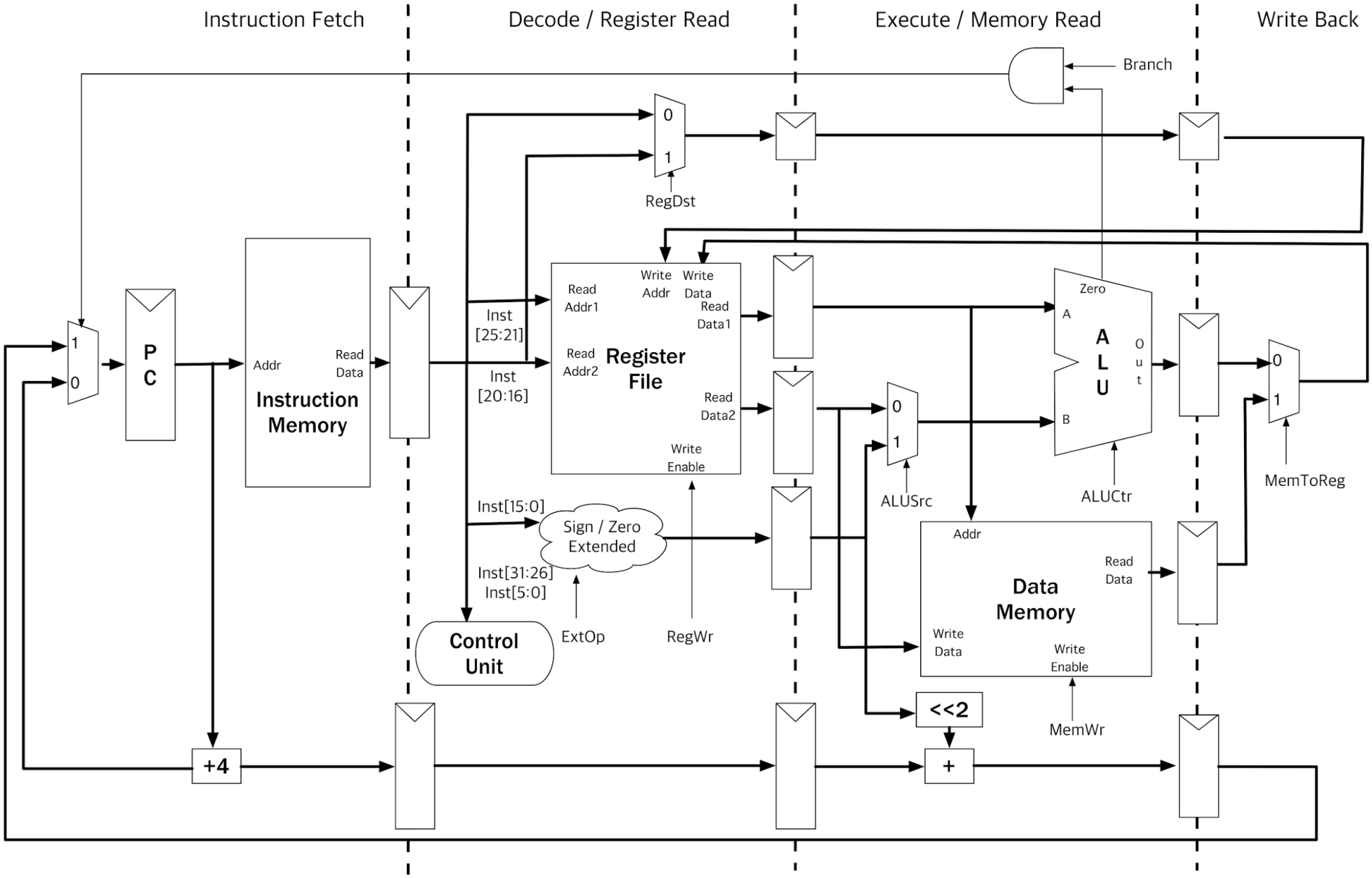


Fill in the control signals:

RegDst	RegWr	nPC_sel	ExtOp	ALUSrc	ALUCtr	MemWr	Mem2Reg	add3
1	1	+4	X	0	add	0	0	1

Pipelining

- The 5 stage pipeline
- Calculating pipelined performance
- Data and control hazards from a naive pipelined CPU
- Ways to reduce stalls, including:
  - Forwarding
  - Forward comparator
  - Delay slots
  - Branch prediction


Instruction Fetch
Decode / Register Read
Execute / Memory Read
Write Back
Branch
RegDst
PC
Addr
Read Data
Instruction Memory
Inst [25:21]
Inst [20:16]
Inst[15:0]
Inst[31:26]
Inst[5:0]
Sign / Zero Extended
ExtOp
Control Unit
Read Addr1
Read Addr2
Write Addr
Write Data
Read Data1
Read Data2
Register File
Write Enable
RegWr
Zero
A
B
ALU
Out
ALUSrc
ALUCtr
Addr
Read Data
Data Memory
Write Data
Write Enable
MemWr
MemToReg
<<2
+
+4

How many stalls?

How many stalls are needed for the code below without/with forwarding?

```
addiu $s0, $t0, 1
xor $s1, $s0, $t1
```

What about the following?

```
lw $s0, 0($t0)
xor $s1, $s0, $t1
```

How many stalls?

How many stalls are needed for the code below without/with forwarding? 2 / 0

```
addiu $s0, $t0, 1
xor $s1, $s0, $t1
```

What about the following? 2 / 1

```
lw $s0, 0($t0)
xor $s1, $s0, $t1
```

How many stalls?

Consider each separately. How many stalls does a branch/jump need on:

- a naive pipelined CPU (no optimization)
- a CPU w/ forward comparator
- a CPU with branch prediction of never take branch

How many stalls?

Consider each separately. How many stalls does a branch/jump need on:

- a naive pipelined CPU (no optimization)

  branch: 2, jump: 1

- a CPU w/ forward comparator

  branch: 1, jump: 1

- a CPU with branch prediction of never take branch

  branch: 0 or 2, jump: 1

Triple Add: Pipelining

For the code on the next slide, calculate the number of stalls needed assuming that the 5-stage CPU has:

- no forwarding, no forward comparator, no delay slots (naive CPU)
- forwarding only
- forwarding + forward comparator + delay slots

Triple Add: Pipelining

```
LOOP:   lw $t0, 0($a0)
        ori $t0, $t0, 0xFFFF
        add3 $t0, $t1, $a1
        sw $t0, 0($a0)
        addiu $a0, $a0, 4
        addiu $t1, $t1, 1
        bne $a0, $a2, LOOP
```

Triple Add: Pipelining

Naive CPU: (data dependencies in blue)

```
LOOP:   lw $t0, 0($a0)          # 2 (data)
        ori $t0, $t0, 0xFFFF    # 2 (data)
        add3 $t0, $t1, $a1      # 2 (data)
        sw $t0, 0($a0)
        addiu $a0, $a0, 4       # 1 (data)
        addiu $t1, $t1, 1
        bne $a0, $a2, LOOP    # 2 (control)
```

Triple Add: Pipelining

With forwarding:

```
LOOP:   lw $t0, 0($a0)        # 1 (load-use)
        ori $t0, $t0, 0xFFFF
        add3 $t0, $t1, $a1
        sw $t0, 0($a0)
        addiu $a0, $a0, 4
        addiu $t1, $t1, 1
        bne $a0, $a2, LOOP    # 2 (control)
```

Triple Add: Pipelining

Forwarding + forward comparator + delay slots:

```
LOOP:   lw $t0, 0($a0)          # 1 (load-use)
        ori $t0, $t0, 0xFFFF
        add3 $t0, $t1, $a1
        sw $t0, 0($a0)
        addiu $a0, $a0, 4
        bne $a0, $a2, LOOP
        addiu $t1, $t1, 1       # delay slot
```

Caches & AMAT

Martin Maas

What is a Cache?

- Fast memory near the CPU.
- Stores memory in units of “cache blocks”.
- Cache blocks are aligned in memory, e.g., Block 1: [0,32[, Block 2: [32,64[,...
- Memory accesses go to the cache first, checks if the block with the address is there already, and only if not goes to memory.
- Cache provides a set of slots that can each hold a specific cache block.

Fully-associative Cache

- Most intuitive way to design a cache.
- Treat cache as collection of blocks and when full, always replace the least-recently used (LRU) one; or pick based on other policy.
- When checking for a block, it could be anywhere -- need to search in each slot.
- Very expensive to implement in hardware, need to compare against every slot.
- Large hit time, not feasible if large capacity.

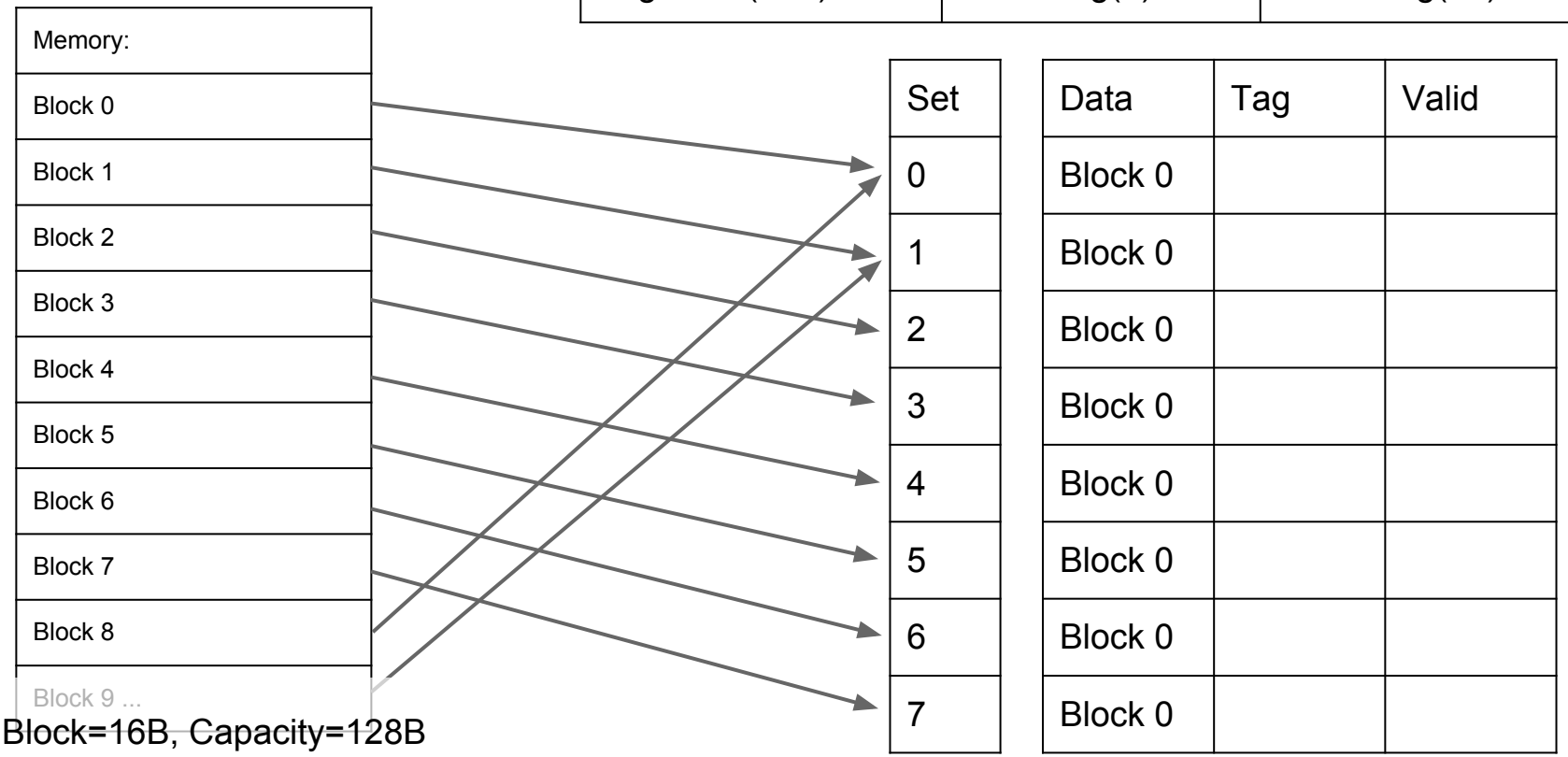
Direct-mapped Cache

- Second attempt: Could avoid checking every slot by making blocks go to exactly one slot.
- Fast: only need to check one cache entry.
- How to decide? Split up the address:

Tag: Addr Space Size - (#I + #O)	Index: log(Number of sets)	Offset: log(Block size (bytes))

Direct-mapped Cache

Tag: Addr Space Size - (#I + #O)	Index: log(Number of sets)	Offset: log(Block size (bytes))
Tag: 32 - (3+4) = 25	Index: log(8) = 3	Offset: log(16) = 4



Block=16B, Capacity=128B

N-way Set-Associativity

- Problem with direct-mapped: Lots of conflicts from blocks mapping to the same set.
- Get the best of both worlds with N-way set associativity: Divide cache into “sets” where the address tells you which set to go to, and then within the set, be associative.
- N tells you how many slots (“ways”) per set. That’s the number of entries you need to compare for each memory access.

Set-associative Cache

Tag: Addr Space Size - (#I + #O)	Index: log(Number of sets)	Offset: log(Block size (bytes))

Tag: 32 - (3+4) = 25	Index: log(8) = 3	Offset: log(16) = 4

Set				
0	Block 0	Block 1	Block 2	Block 3
1	Block 0	Block 1	Block 2	Block 3
2	Block 0	Block 1	Block 2	Block 3
3	Block 0	Block 1	Block 2	Block 3
4	Block 0	Block 1	Block 2	Block 3
5	Block 0	Block 1	Block 2	Block 3
6	Block 0	Block 1	Block 2	Block 3
7	Block 0	Block 1	Block 2	Block 3

Block=16B, Capacity=512B

Trade-offs

- Higher associativity = lower miss rate (fewer conflicts), higher hit time (more complexity).
- Direct mapped: Associativity of 1, Fully associative: Associativity of <Number of slots in the cache>
- Number of tag/index/offset bits depends on block size and number of sets.
- Number of sets = capacity / size per set = capacity / (block size * N)

In a direct mapped cache, the number of blocks in the cache is always the same as:

A) The number of bytes in the cache

B) The number of offset bits

C) The number of sets

D) The number of rows

E) The number of valid bits

F) 2^(The number of index bits)

G) The number of index bits

(more than one may be correct)

Warmup Question

In a direct mapped cache, the number of blocks in the cache is always the same as:

A) The number of bytes in the cache

B) The number of offset bits

C) The number of sets

D) The number of rows

E) The number of valid bits

F) 2^(The number of index bits)

G) The number of index bits

(more than one may be correct)

Warmup Question

We have a standard 32-bit byte-addressed MIPS machine with 4 GiB RAM, a 4-way set-associative CPU data cache that uses 32 byte blocks, a LRU replacement policy, and has a total capacity of 16 KiB. Consider the following C code and answer the questions below.

```
#define SIZE_OF_A 2048

typedef struct {
      int x;
      int y[3];
} node;

int count_x(node *A, int x) {
      int k = 0;
      for (int i = 0; i < SIZE_OF_A; i++)
            if (A[i].x == x) {
                  k++;
            }
      return k;
}
```

a) How many bits are used for the tag, index, and offset?

Tag	Index	Offset
20	7	5

b) We call `count_x` with all values of `x` from 0 to 65535 to count the number of times that each `x` occurs in `A`. The value of `A` is the same in every call. The cache is cold at the beginning of execution. What is the cache hit rate?

50%

Example Question: Spring '14, Final, M2

We have a standard 32-bit byte-addressed MIPS machine with 4 GiB RAM, a 4-way set-associative CPU data cache that uses 32 byte blocks, a LRU replacement policy, and has a total capacity of 16 KiB. Consider the following C code and answer the questions below.

```
#define SIZE_OF_A 2048

typedef struct {
      int x;
      int y[3];
} node;

int count_x(node *A, int x) {
      int k = 0;
      for (int i = 0; i < SIZE_OF_A; i++)
            if (A[i].x == x) {
                  k++;
            }
      return k;
}
```

c) Let's say that we increase our CPU cache associativity to 8-way.
What is our cache hit rate now?

50%

d) What would be the approximate cache rate if we changed our CPU cache to use a *Most Recently Used* (MRU) cache replacement policy, and we change the cache to be *fully associative*?

75%

Example Question: Spring '14, Final, M2

Types of Cache Misses

- 1) Compulsory: The first time you bring data into the cache.
- 2) Conflict: They would not have happened with a fully-associative cache.
- 3) Capacity: They would not have happened with an infinitely large cache.
- Finding out which one it is: Check 1, 2, 3 in this order and take the first one that applies.

Given a direct-mapped cache, initially empty, and the following memory access pattern (all byte addresses/accesses, 32-bit addresses)

8, 0, 5, 32, 0, 42, 9

Of what kinds are the different cache misses, and what blocks are in the cache after these accesses if the cache has a capacity of 16B?

Practice Question

Capacity: 16B, Block size: 4B

First, need T:I:O

Offset = log2(block size) = log2(4) = 2

Index = log2(#slots) = log2(4) = 2

Tag = 32 - 2 - 2 = 28

Say M[x] for the memory at address x

Practice Question

T:I:O = 28:2:2, Accesses: 8, 0, 5, 32, 0, 42, 9

	Byte 3	Byte 2	Byte 1	Byte 0
Index 0				
Index 1				
Index 2				
Index 3				

Practice Question

T:I:O = 28:2:2, Accesses: 8, 0, 5, 32, 0, 42, 9

	Byte 3	Byte 2	Byte 1	Byte 0
Index 0				
Index 1				
Index 2	M[11]	M[10]	M[9]	M[8]
Index 3				

Addr 8 = 0b1000: Tag = 0b0, Index = 0b10, Offset = 0b00

Compulsory Miss

Practice Question

T:I:O = 28:2:2, Accesses: 8, 0, 5, 32, 0, 42, 9

	Byte 3	Byte 2	Byte 1	Byte 0
Index 0	M[3]	M[2]	M[1]	M[0]
Index 1				
Index 2	M[11]	M[10]	M[9]	M[8]
Index 3				

Addr 0 = 0b0000: Tag = 0b0, Index = 0b00, Offset = 0b00

Compulsory Miss

Practice Question

T:I:O = 28:2:2, Accesses: 8, 0, 5, 32, 0, 42, 9

	Byte 3	Byte 2	Byte 1	Byte 0
Index 0	M[3]	M[2]	M[1]	M[0]
Index 1	M[7]	M[6]	M[5]	M[4]
Index 2	M[11]	M[10]	M[9]	M[8]
Index 3				

Addr 5 = 0b0101: Tag = 0b0, Index = 0b01, Offset = 0b01

Compulsory Miss

Practice Question

T:I:O = 28:2:2, Accesses: 8, 0, 5, 32, 0, 42, 9

	Byte 3	Byte 2	Byte 1	Byte 0
Index 0	M[35]	M[34]	M[33]	M[32]
Index 1	M[7]	M[6]	M[5]	M[4]
Index 2	M[11]	M[10]	M[9]	M[8]
Index 3				

Addr 32 = 0b100000: Tag = 0b10, Index = 0b00, Offset = 0b00

Compulsory Miss

Practice Question

T:I:O = 28:2:2, Accesses: 8, 0, 5, 32, 0, 42, 9

	Byte 3	Byte 2	Byte 1	Byte 0
Index 0	M[3]	M[2]	M[1]	M[0]
Index 1	M[7]	M[6]	M[5]	M[4]
Index 2	M[11]	M[10]	M[9]	M[8]
Index 3				

Addr 0 = 0b0000: Tag = 0b0, Index = 0b00, Offset = 0b00

Conflict Miss

Practice Question

T:I:O = 28:2:2, Accesses: 8, 0, 5, 32, 0, 42, 9

	Byte 3	Byte 2	Byte 1	Byte 0
Index 0	M[3]	M[2]	M[1]	M[0]
Index 1	M[7]	M[6]	M[5]	M[4]
Index 2	M[43]	M[42]	M[41]	M[40]
Index 3				

Addr 42 = 0b101010: Tag = 0b10, Index = 0b10, Offset = 0b10

Compulsory Miss

Practice Question

T:I:O = 28:2:2, Accesses: 8, 0, 5, 32, 0, 42, 9

	Byte 3	Byte 2	Byte 1	Byte 0
Index 0	M[3]	M[2]	M[1]	M[0]
Index 1	M[39]	M[38]	M[37]	M[36]
Index 2	M[11]	M[10]	M[9]	M[8]
Index 3				

Addr 9 = 0b1001: Tag = 0b0, Index = 0b10, Offset = 0b01

Capacity Miss

Practice Question

This C code runs on a 32-bit MIPS machine with 4 GiB of memory and a single L1 cache. Vectors `A,B` live in different places of memory, are of equal size (`n` is a power of 2 and a [natural number] multiple of the cache size), block aligned. The size of the cache is C, a power of 2 (and always bigger than the block size, obviously).

```
// sizeof(uint8_t) = 1
SwapLeft(uint8_t *A, uint8_t *B, int n) {
   uint8_t tmp;
   for (int i = 0; i < n; i++) {
      tmp = A[i];
      A[i] = B[i];
      B[i] = tmp;
   }
}
```

```
// sizeof(uint8_t) = 1
SwapRight(uint8_t *A, uint8_t *B, int n) {
   uint8_t tmpA, tmpB;
   for (int i = 0; i < n; i++) {

      _____

      _____

      _____

      _____

   }
}
```

Let's first just consider the `SwapLeft` code for parts (a) and (b).

a) If the cache is **direct mapped** and the *best* hit:miss ratio is "H:1", what is the block size in bytes? ____ **(H+1)/2**

b) What is the *worst* hit:miss *ratio*? ____ **0:1** ____

Example Question: Spring '13, Final, M2

This C code runs on a 32-bit MIPS machine with 4 GiB of memory and a single L1 cache. Vectors **A,B** live in different places of memory, are of equal size (**n** is a power of 2 and a [natural number] multiple of the cache size), block aligned. The size of the cache is C, a power of 2 (and always bigger than the block size, obviously).

```
// sizeof(uint8_t) = 1
SwapLeft(uint8_t *A, uint8_t *B, int n) {
   uint8_t tmp;
   for (int i = 0; i < n; i++) {
      tmp = A[i];
      A[i] = B[i];
      B[i] = tmp;
   }
}
```

```
// sizeof(uint8_t) = 1
SwapRight(uint8_t *A, uint8_t *B, int n) {
   uint8_t tmpA, tmpB;
   for (int i = 0; i < n; i++) {
      uint8_t tmpA = A[i];
      uint8_t tmpB = B[i];
      B[i] = tmpA;
      A[i] = tmpB;
   }
}
```

c) Fill in the code for **SwapRight** so that it does the same thing as **SwapLeft** but improves the (b) hit:miss ratio. You may not need all the blanks.

d) If the block size (in bytes) is *a*, what is the *worst* hit:miss ratio for **SwapRight**? 2a-1:2a+1

Example Question: Spring '13, Final, M2

This C code runs on a 32-bit MIPS machine with 4 GiB of memory and a single L1 cache. Vectors `A,B` live in different places of memory, are of equal size (`n` is a power of 2 and a [natural number] multiple of the cache size), block aligned. The size of the cache is C, a power of 2 (and always bigger than the block size, obviously).

```
// sizeof(uint8_t) = 1
SwapLeft(uint8_t *A, uint8_t *B, int n) {
   uint8_t tmp;
   for (int i = 0; i < n; i++) {
      tmp = A[i];
      A[i] = B[i];
      B[i] = tmp;
   }
}
```

```
// sizeof(uint8_t) = 1
SwapRight(uint8_t *A, uint8_t *B, int n) {
   uint8_t tmpA, tmpB;
   for (int i = 0; i < n; i++) {
      uint8_t tmpA = A[i];
      uint8_t tmpB = B[i];
      B[i] = tmpA;
      A[i] = tmpB;
   }
}
```

e) We next change the cache to be **2-way set-associative**, and let's go back to just considering `SwapLeft`. What is the **worst** hit:miss ratio for `SwapLeft` with the following replacement policies? The cache size is *C* (bytes), the block size is *a* (bytes), LRU = Least Recently Used, MRU = Most Recently Used.

LRU and **an empty cache**	**MRU** and **a full cache**
2a-1:1	0:1

Example Question: Spring '13, Final, M2

Cache Miss Rates

- Local Miss Rate: Fraction of accesses going into a cache that misses.
- Global Miss Rate: Fraction of all accesses that miss at this level and all levels below.
- For inclusive caches, global miss rate is usually easier to determine.
- L1: global and local miss rate are the same
- Otherwise: LocalN = GlobalN / Global(N-1)

AMAT

- Estimate efficiency of memory hierarchy.
- Approach 1:

  AMAT = L1 Hit Time + L1 Miss Rate * L1 Miss Penalty
  = L1 Hit Time + L1 Miss Rate * L2 AMAT
  = L1 Hit Time + L1 Miss Rate * (L2 Hit Time +
  L2 Miss Rate * L2 Miss Penalty) = …
- Approach 2:

  AMAT = L1 Hit Time + L1 Miss Rate * L2 Hit Time
  + Global L2 Miss Rate * L3 Hit Time
  + Global L3 Miss Rate * (...) + ...

(b) Given the following specification:

For every 1000 CPU-to-memory references
- 40 will miss in L1$;
- 20 will miss in L2$;
- 10 will miss in L3$;

L1$ hits in 1 clock cycle;
L2$ hits in 10 clock cycles;
L3$ hits in 100 clock cycles;
Main memory access is 400 clock cycles;
There are 1.25 memory references per instruction; and
The ideal CPI is 1.

Answer the following questions:

(i) What is the local miss rate in the L2$?

20/40 = 50%

(ii) What is the global miss rate in the L2$?

20/1000 = 2%

(iii) What is the local miss rate in the L3$?

10/20 = 50%

(iv) What is the global miss rate in the L3$?

10/1000 = 1%

Example Question: Fall '10, Final, Q4

(b) Given the following specification:

For every 1000 CPU-to-memory references
- 40 will miss in L1$;
- 20 will miss in L2$;
- 10 will miss in L3$;

L1$ hits in 1 clock cycle;
L2$ hits in 10 clock cycles;
L3$ hits in 100 clock cycles;
Main memory access is 400 clock cycles;
There are 1.25 memory references per instruction; and
The ideal CPI is 1.

Answer the following questions:

(v) What is the AMAT with all three levels of cache?

1 + 4%*(10+50%*(100+50%*400)) = 1 + 0.4 + 2%*300 = 7.4

(vi) What is the AMAT for a *two-level* cache *without L3$?*

1 + 4%*10 + 2%*400 = 1 + 0.4 + 8 = 9.4

Example Question: Fall '10, Final, Q4

GOOD LUCK!!!