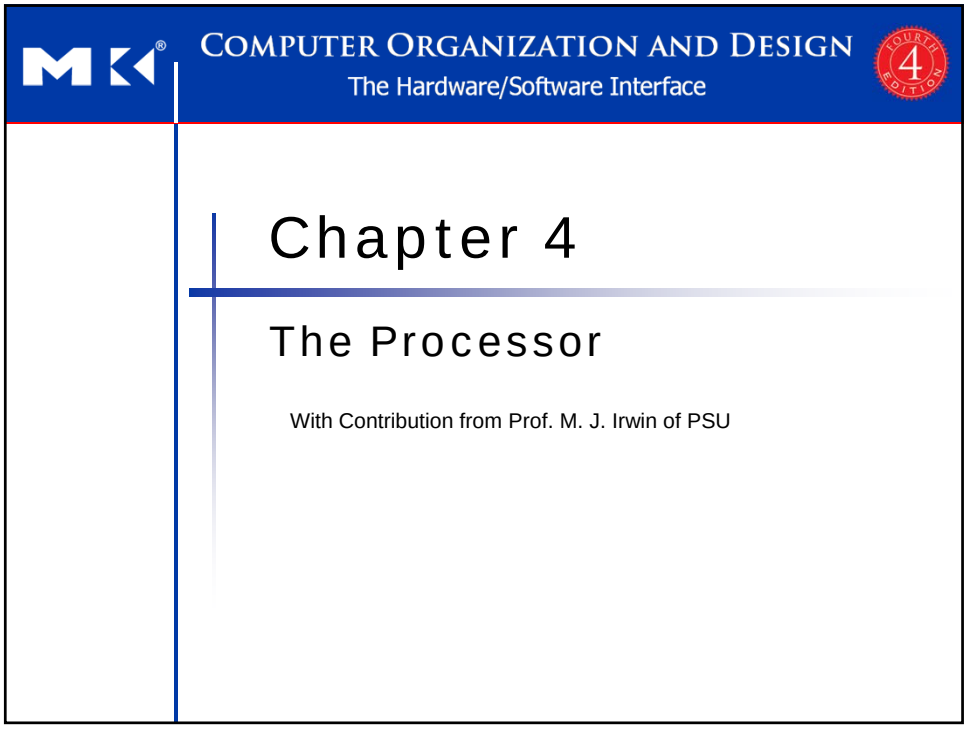


Chapter Roadmap

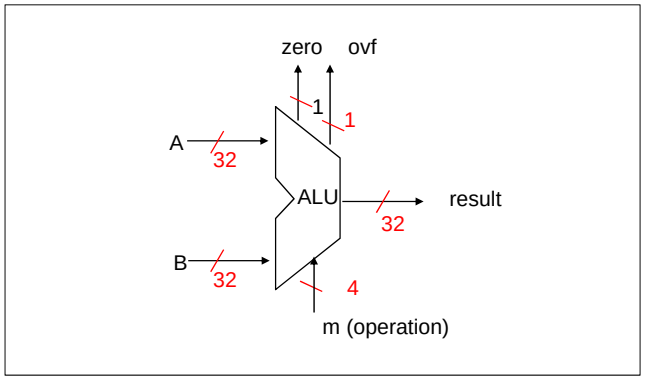
- How to design an ALU?
- MIPS datapath
- Pipelining
- Hazards
- Real stuff

Introduction

- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware
- We will examine two MIPS implementations
  - A simplified version
  - A more realistic pipelined version
- Simple subset, shows most aspects
  - Memory reference: `lw`, `sw`
  - Arithmetic/logical: `add`, `sub`, `and`, `or`, `slt`
  - Control transfer: `beq`, `j`

Design of a Simple ALU

- 2 inputs, operation and an output



Possible Representations

Sign Mag.	Two's Comp.	One's Comp.
	1000 = -8	
1111 = -7	1001= -7	1000 = -7
1110 = -6	1010 = -6	1001 = -6
1101 = -5	1011 = -5	1010 = -5
1100 = -4	1100 = -4	1011 = -4
1011 = -3	1101 = -3	1100 = -3
1010 = -2	1110 = -2	1101 = -2
1001 = -1	1111 = -1	1110 = -1
1000 = -0		1111 = -0
0000 = +0	0000 = 0	0000 = +0
0001 = +1	0001 = +1	0001 = +1
0010 = +2	0010 = +2	0010 = +2
0011 = +3	0011 = +3	0011 = +3
0100 = +4	0100 = +4	0100 = +4
0101 = +5	0101 = +5	0101 = +5
0110 = +6	0110 = +6	0110 = +6
0111 = +7	0111 = +7	0111 = +7

- Issues:
  - balance
  - number of zeros
  - ease of operations
- Which one is best? Why?

MIPS Representations

- 32-bit signed numbers (2's complement):

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +\ 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = +\ 2_{ten}$
...

maxint

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +\ 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -\ 2{,}147{,}483{,}646_{ten}$
...

minint

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -\ 3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -\ 2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -\ 1_{ten}$

- What if the bit string represented addresses?
  - need operations that also deal with only positive (unsigned) integers

Two's Complement Operations

- Negating a two's complement number – complement all the bits and then add a 1
  - remember: "negate" and "invert" are quite different!
  - Starting from LSb, all 0's as is, first 1 as is, then invert
- Converting n-bit numbers into numbers with more than n bits:
  - MIPS 16-bit immediate gets converted to 32 bits for arithmetic
  - sign extend - copy the most significant bit (the sign bit) into the other bits

```
0010  -> 0000 0010
1010  -> 1111 1010
```

  - sign extension versus zero extend (`lb` vs. `lbu`)

Addition & Subtraction

- Just like in grade school (carry/borrow 1s)

```
  0111          0111          0110
+ 0110        - 0110        - 0101
  1101          0001          0001
```

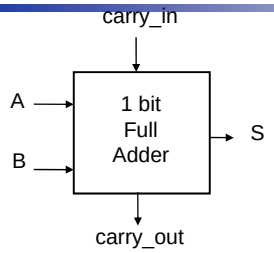
- Two's complement operations are easy
  - do subtraction by negating and then adding

```
  0111    →     0111
- 0110    →   + 1010
  0001        1 0001
```

- Overflow (result too large for finite computer word)
  - e.g., adding two n-bit numbers does not yield an n-bit number

```
  0111
+ 0001
 1000
```

Building a 1-bit Binary Adder

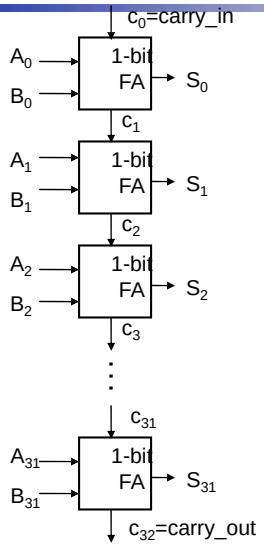


A	B	carry_in	carry_out	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

S = A xor B xor carry_in
carry_out = A&B | A&carry_in | B&carry_in
(majority function)

- How can we use it to build a 32-bit adder?
- How can we modify it easily to build an adder/subtractor?

Building 32-bit Adder



- Just connect the carry-out of the least significant bit FA to the carry-in of the next least significant bit and connect . . .
- Ripple Carry Adder (RCA)
  - advantage: simple logic, so small (low cost)
  - disadvantage: slow and lots of glitching (so lots of energy consumption)

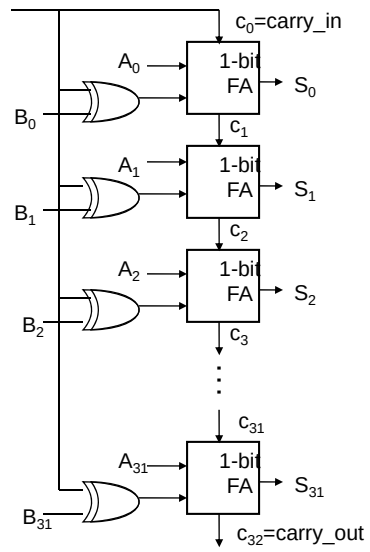
A 32-bit Ripple Carry Adder/Subtractor

- Remember 2's complement is just
  - complement all the bits

control (0=add,1=sub), B_0 → B_0 if control = 0, $!B_0$ if control = 1

  - add a 1 in the least significant bit

```
A    0111   →    0111
B  - 0110   →  +
```



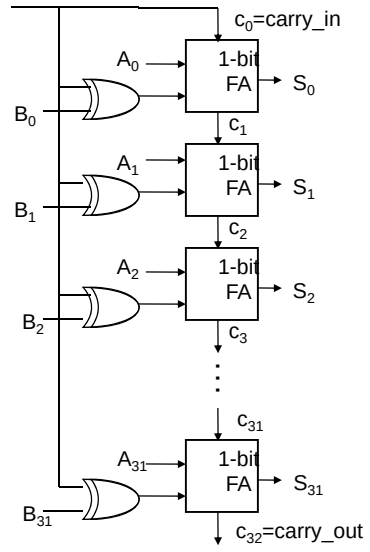
A 32-bit Ripple Carry Adder/Subtractor

- Remember 2's complement is just
  - complement all the bits

control (0=add,1=sub), B_0 → B_0 if control = 0, $!B_0$ if control = 1

  - add a 1 in the least significant bit

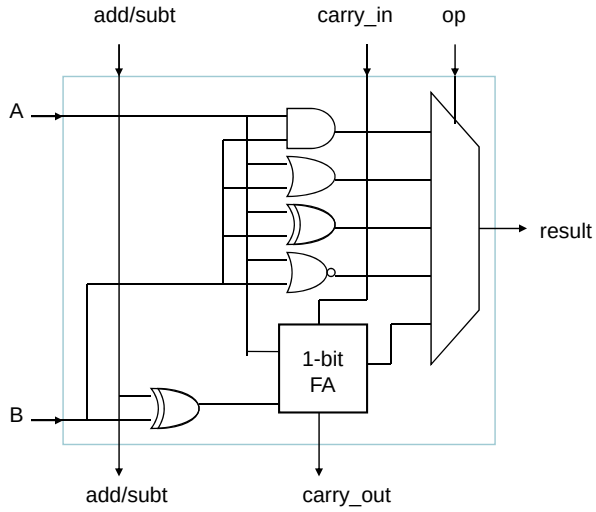
```
A    0111   →    0111
B  - 0110   →  + 1001
  ------          1
  0001          ------
                1 0001
```

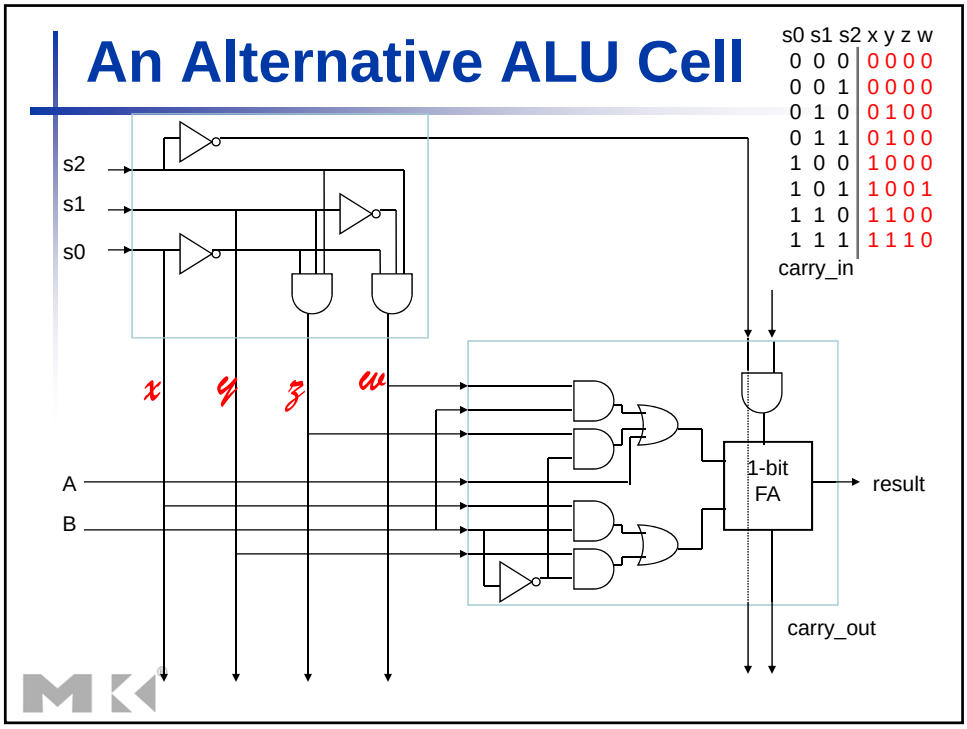


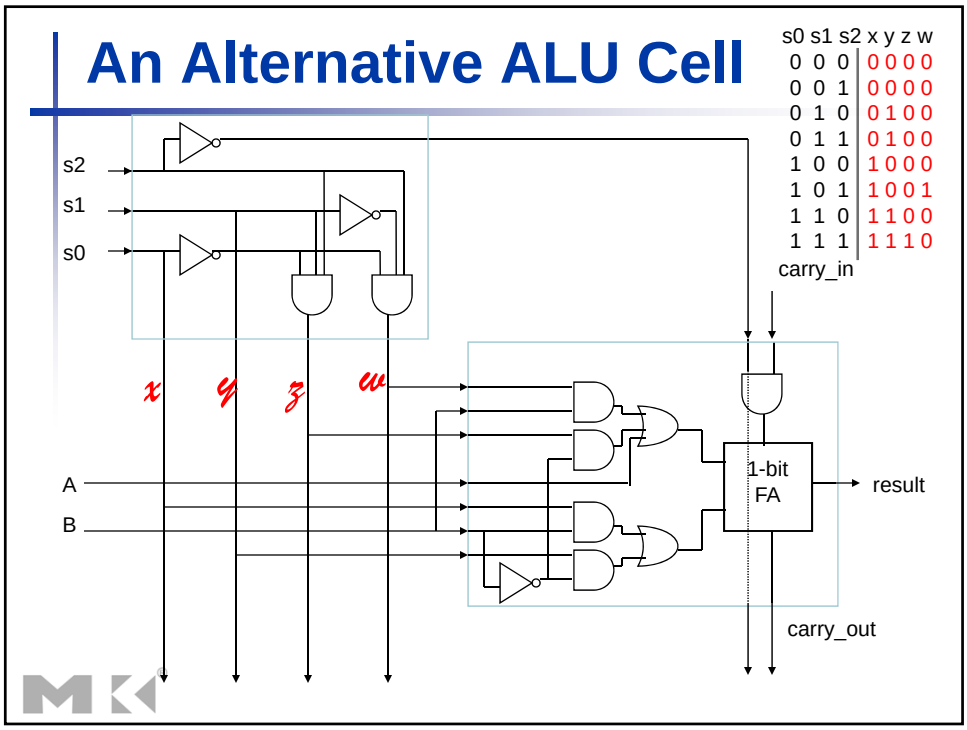
Overflow Detection and Effects

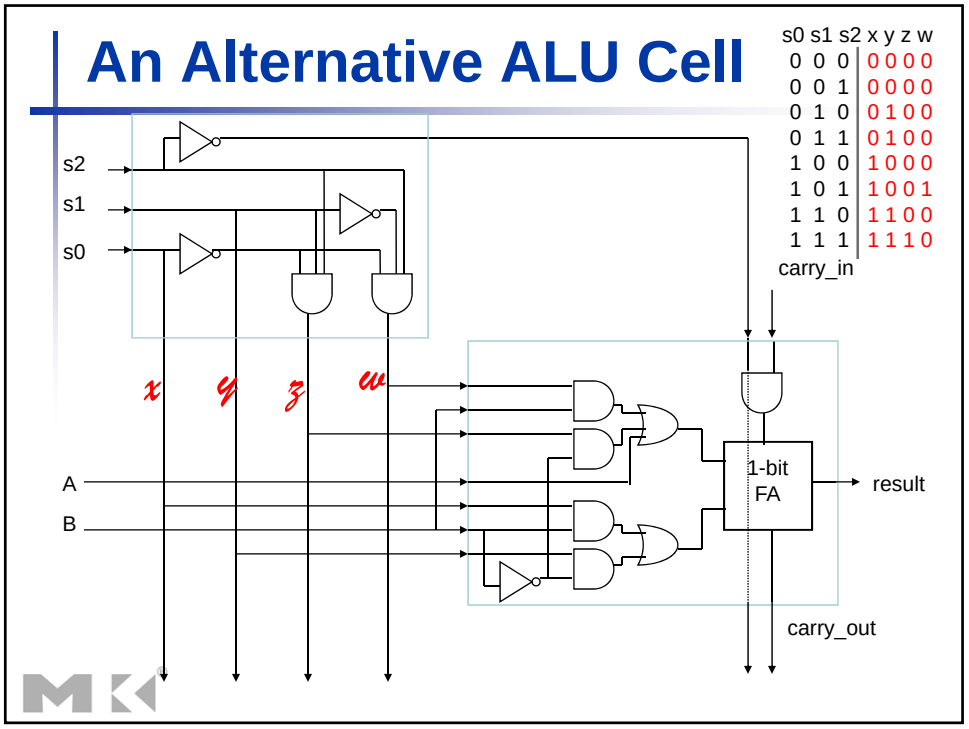
- Overflow: the result is too large to represent in the number of bits allocated
- When adding operands with different signs, overflow cannot occur! Overflow occurs when
  - adding two positives yields a negative
  - or, adding two negatives gives a positive
  - or, subtract a negative from a positive gives a negative
  - or, subtract a positive from a negative gives a positive
- On overflow, an exception (interrupt) occurs
  - Control jumps to predefined address for exception
  - Interrupted address (address of instruction causing the overflow) is saved for possible resumption
- Don't always want to detect (interrupt on) overflow

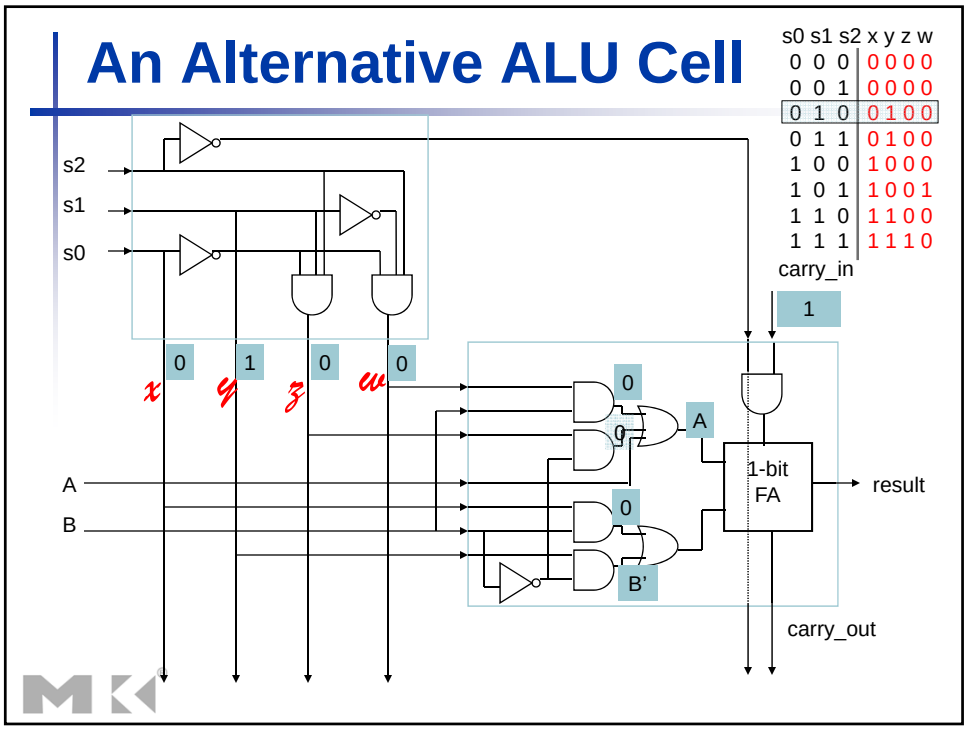
A Simple ALU Cell with Logic Op Support

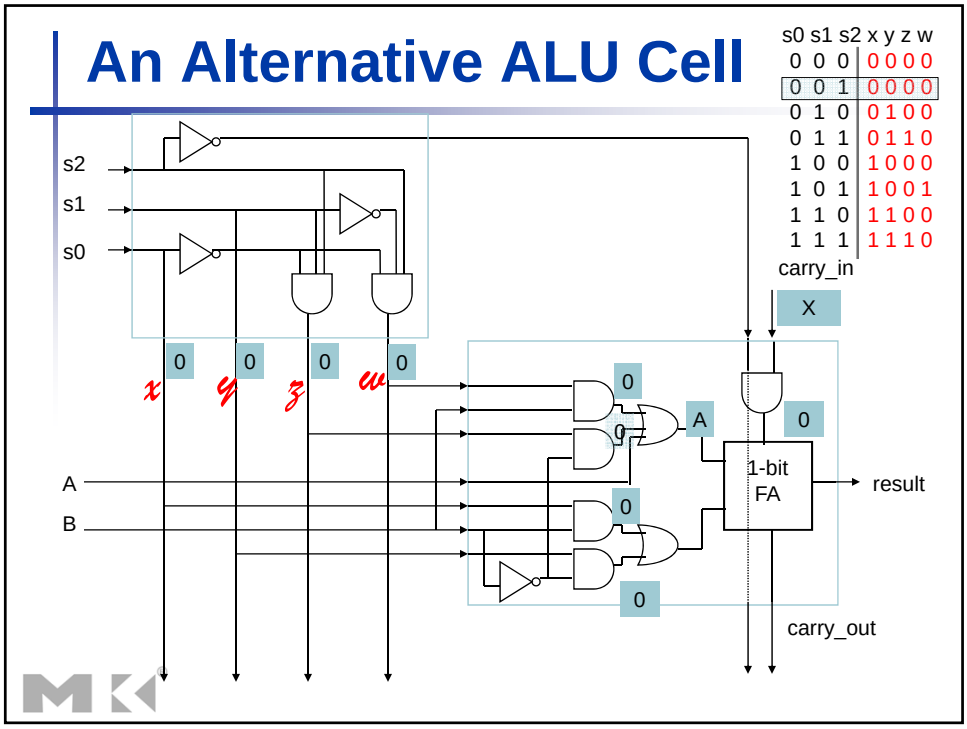


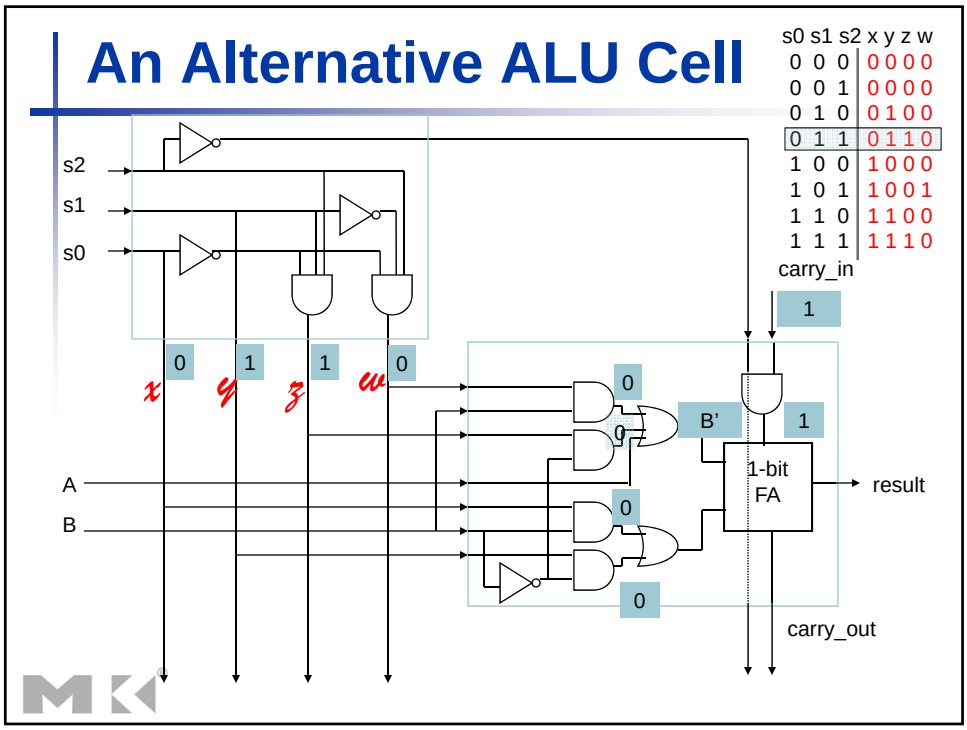

An Alternative ALU Cell
s0 s1 s2 x y z w
0 0 0 0 0 0 0
0 0 1 0 0 0 0
0 1 0 0 1 0 0
0 1 1 0 1 0 0
1 0 0 1 0 0 0
1 0 1 1 0 0 1
1 1 0 1 1 0 0
1 1 1 1 1 1 0
carry_in
s2
s1
s0
x
y
z
w
A
B
1-bit FA
result
carry_out


An Alternative ALU Cell
s0 s1 s2 x y z w
0 0 0 0 0 0 0
0 0 1 0 0 0 0
0 1 0 0 1 0 0
0 1 1 0 1 0 0
1 0 0 1 0 0 0
1 0 1 1 0 0 1
1 1 0 1 1 0 0
1 1 1 1 1 1 0
carry_in
s2
s1
s0
x
y
z
w
A
B
1-bit FA
result
carry_out


An Alternative ALU Cell
s0 s1 s2 x y z w
0 0 0 0 0 0 0
0 0 1 0 0 0 0
0 1 0 0 1 0 0
0 1 1 0 1 0 0
1 0 0 1 0 0 0
1 0 1 1 0 0 1
1 1 0 1 1 0 0
1 1 1 1 1 1 0
carry_in
s2
s1
s0
x
y
z
w
A
B
1-bit FA
result
carry_out


An Alternative ALU Cell
s0 s1 s2 x y z w
0 0 0 0 0 0 0
0 0 1 0 0 0 0
0 1 0 0 1 0 0
0 1 1 0 1 0 0
1 0 0 1 0 0 0
1 0 1 1 0 0 1
1 1 0 1 1 0 0
1 1 1 1 1 1 0
carry_in
1
s2
s1
s0
0
1
0
0
x
y
z
w
0
0
A
0
B'
A
B
1-bit FA
result
carry_out

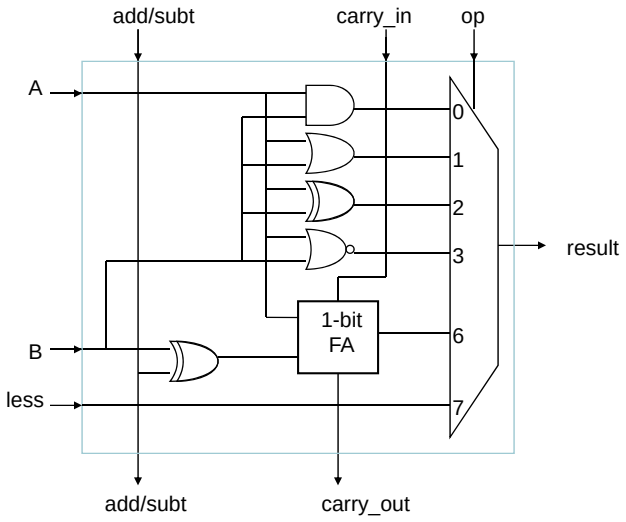

An Alternative ALU Cell
s0 s1 s2 x y z w
0 0 0 0 0 0 0
0 0 1 0 0 0 0
0 1 0 0 1 0 0
0 1 1 0 1 1 0
1 0 0 1 0 0 0
1 0 1 1 0 0 1
1 1 0 1 1 0 0
1 1 1 1 1 1 0
carry_in
X
s2
s1
s0
x
y
z
w
A
B
1-bit FA
result
carry_out


An Alternative ALU Cell
s0 s1 s2 x y z w
0 0 0 0 0 0 0
0 0 1 0 0 0 0
0 1 0 0 1 0 0
0 1 1 0 1 1 0
1 0 0 1 0 0 0
1 0 1 1 0 0 1
1 1 0 1 1 0 0
1 1 1 1 1 1 0
carry_in
1
s2
s1
s0
x
y
z
w
B'
A
B
1-bit FA
result
carry_out

The Alternative ALU Cell's Control Codes

s2	s1	s0	c_in	result	function
0	0	0	0	A	transfer A
0	0	0	1	A + 1	increment A
0	0	1	0	A + B	add
0	0	1	1	A + B + 1	add with carry
0	1	0	0	A – B – 1	subt with borrow
0	1	0	1	A – B	subtract
0	1	1	0	A – 1	decrement A
0	1	1	1	A	transfer A
1	0	0	x	A or B	or
1	0	1	x	A xor B	xor
1	1	0	x	A and B	and
1	1	1	x	!A	complement A

Modifying the ALU Cell for `slt`



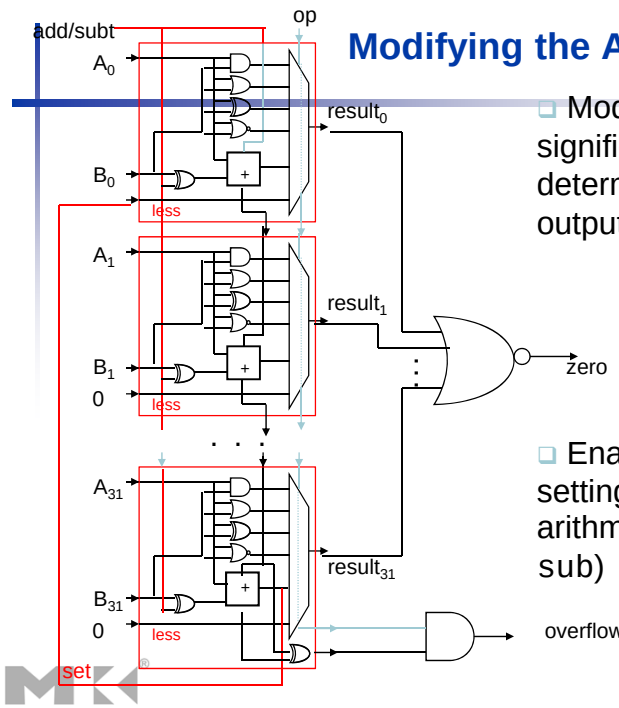
Overflow Detection

- Overflow occurs when the result is too large to represent in the number of bits allocated
  - adding two positives yields a negative
  - or, adding two negatives gives a positive
  - or, subtract a negative from a positive gives a negative
  - or, subtract a positive from a negative gives a positive
- On your own: Prove you can detect overflow by:
  - Carry into MSB xor Carry out of MSB

	0	1	1	1	7
+	0	0	1	1	3

	1	1	0	0	–4
+	1	0	1	1	– 5

Modifying the ALU for Overflow

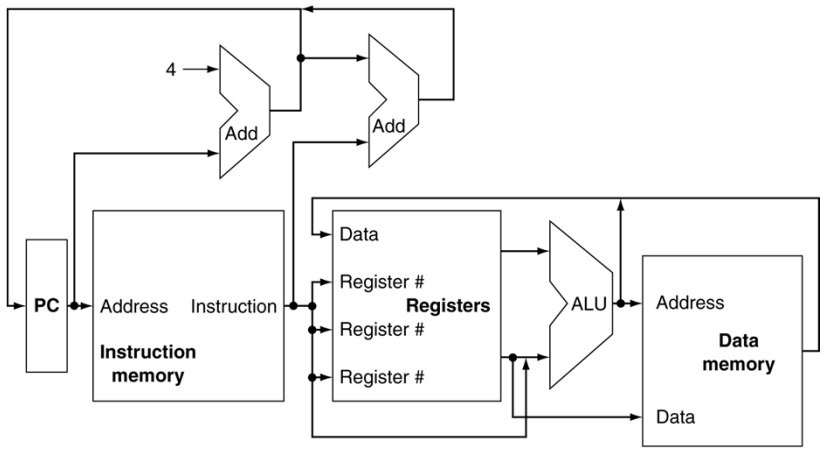


- Modify the most significant cell to determine overflow output setting
- Enable overflow bit setting for signed arithmetic (`add, addi, sub`)

Instruction Execution

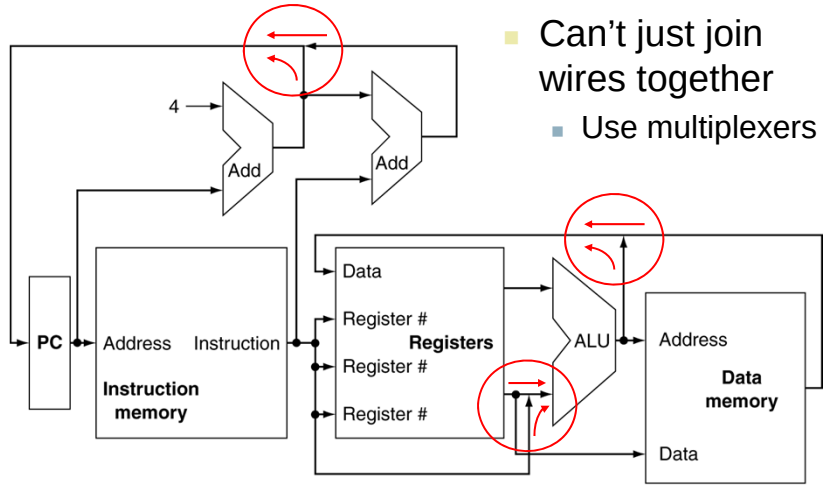
- PC → instruction memory, fetch instruction
- All but jump, need to access data from a register and use the ALU.
- Depending on instruction class
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch target address `beq $r1, $r2, LOC`
  - Access data memory for load/store
  - PC ← target address or PC + 4

CPU Overview

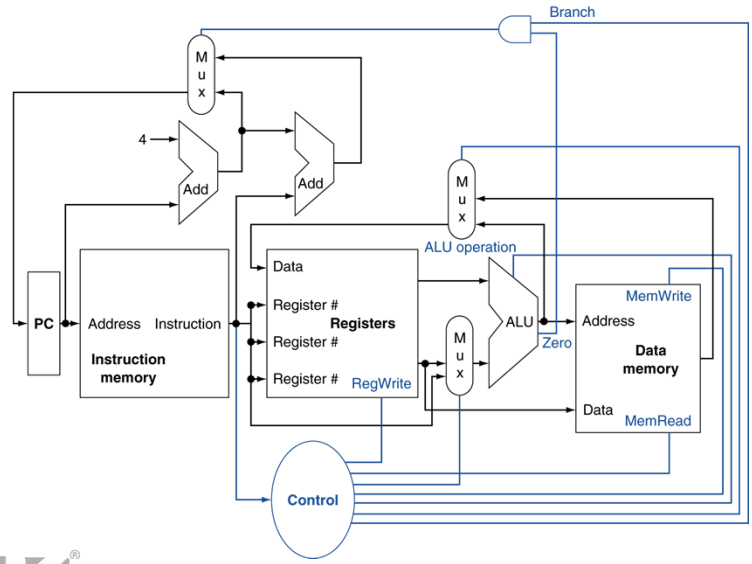


Multiplexers

- Can't just join wires together
  - Use multiplexers



Control

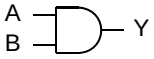


Logic Design Basics

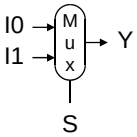
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Combinational Elements

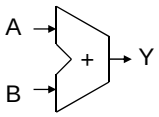
- AND-gate
  - Y = A & B



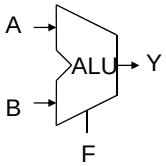
- Multiplexer
  - Y = S ? I1 : I0



- Adder
  - Y = A + B

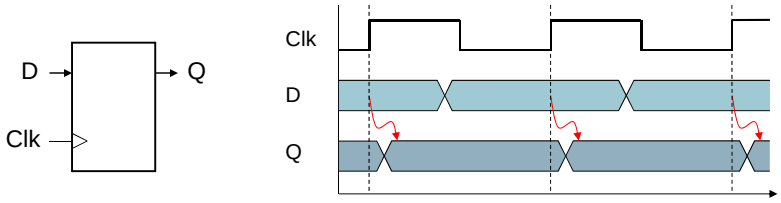


- Arithmetic/Logic Unit
  - Y = F(A, B)



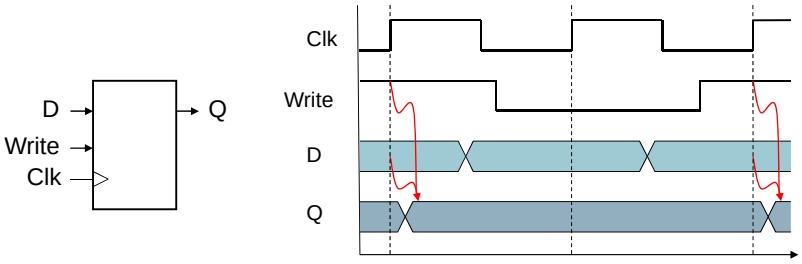
Sequential Elements

- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1



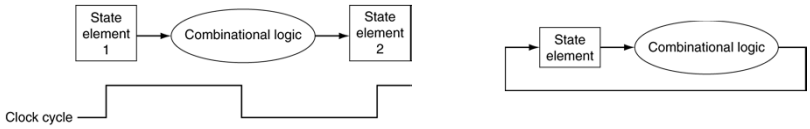
Sequential Elements

- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later



Clocking Methodology

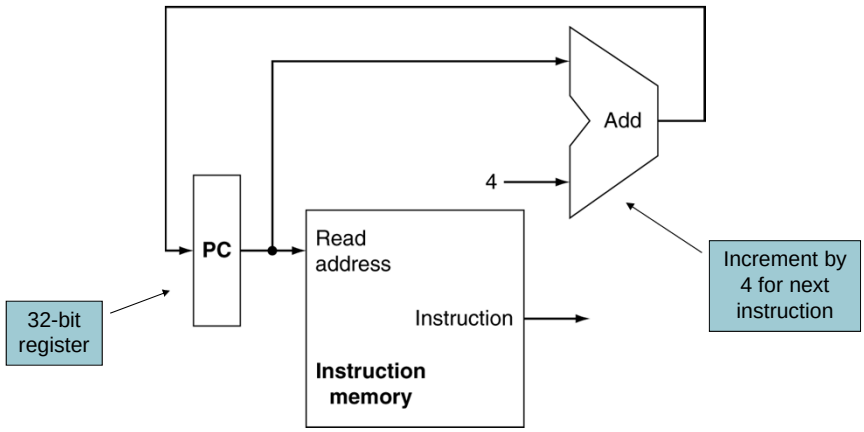
- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period



Building a Datapath

- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, …
- We will build a MIPS datapath incrementally
  - Refining the overview design

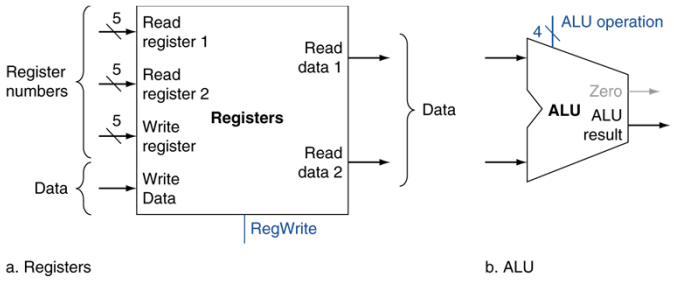
Instruction Fetch



R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result

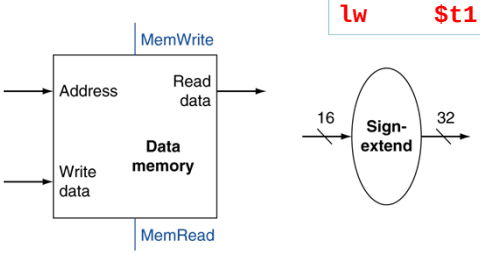
```
add   $t1, $t2, $t3
```



Load/Store Instructions

- Read register operands
- Calculate address using 16-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory

```
lw    $t1, offset($t3)
```



a. Data memory unit b. Sign extension unit

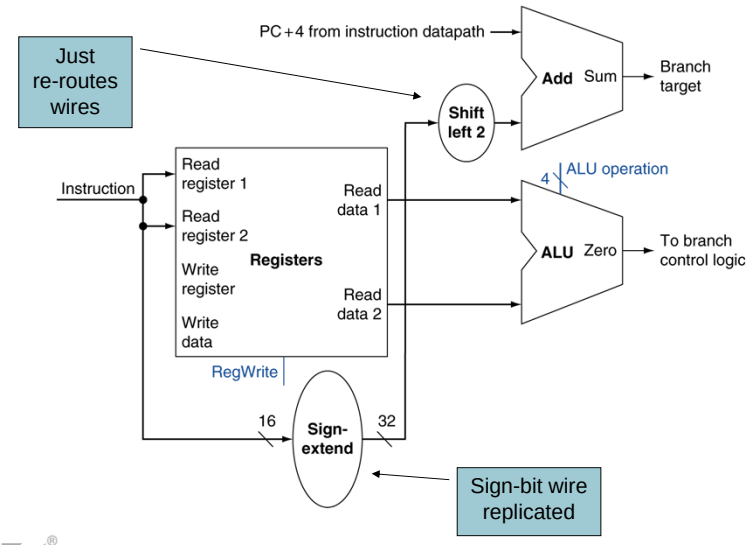
Branch Instructions

- Read register operands
- Compare operands

```
beq    $t1, $t2, offset
```

  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 2 places (word displacement)
  - Add to PC + 4
    - Already calculated by instruction fetch

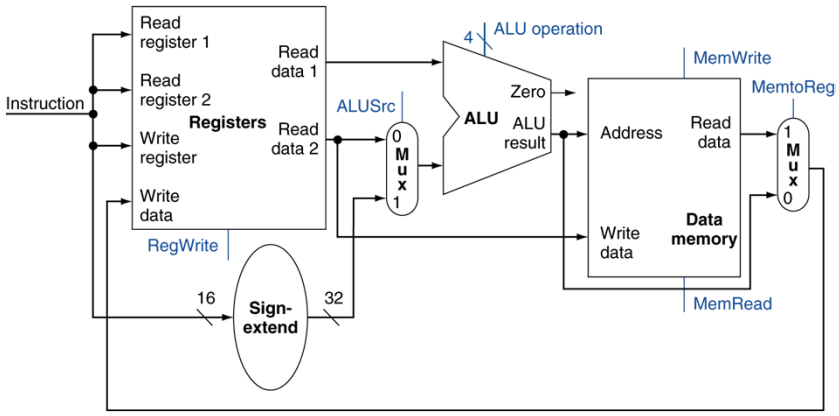
Branch Instructions



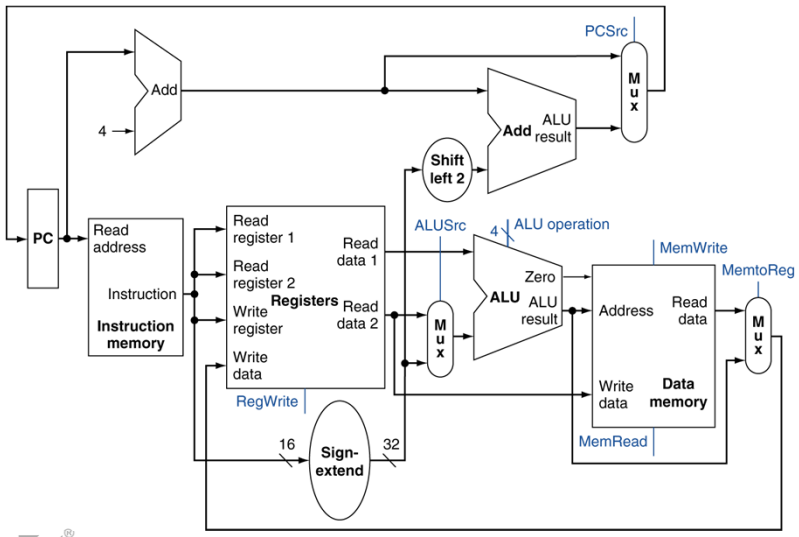
Composing the Elements

- First-cut data path does an instruction in one clock cycle
  - Each datapath element can only do one function at a time
  - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

R-Type/Load/Store Datapath



Full Datapath



ALU Control

- ALU used for
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on funct field

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR