

Processor

Thumrongsak Kosiyatrakul
tkosiyat@cs.pitt.edu

Draft

Learning Objective

How to build a Central Processing Unit (CPU)

Major Questions to Answer


How do we design

- 1 Arithmetic Logic Unit (ALU)
- 2 Datapath
- 3 Control Signals

Do we have any plan?

Our Plan

Using Instructions and their machine languages as a specification:

① 

CORE INSTRUCTION SET			OPCODE / FUNCT (Hex)
NAME, MNEMONIC	FOR- MAT	OPERATION (in Verilog)	
Add	add R	$R[rd] = R[rs] + R[rt]$	(1) 0 / 20 _{hex}
Add Immediate	addi I	$R[rt] = R[rs] + \text{SignExtImm}$	(1,2) 8 _{hex}
Add Imm. Unsigned	addiu I	$R[rt] = R[rs] + \text{SignExtImm}$	(2) 9 _{hex}
Add Unsigned	addu R	$R[rd] = R[rs] + R[rt]$	0 / 21 _{hex}
And	and R	$R[rd] = R[rs] \& R[rt]$	0 / 24 _{hex}
And Immediate	andi I	$R[rt] = R[rs] \& \text{ZeroExtImm}$	(3) 0 _{hex}
Branch On Equal	beq I	if $R[rs] == R[rt]$ PC ← PC + 4 + BranchAddr	(4) 4 _{hex}
Branch On Not Equal	bne I	if $R[rs] != R[rt]$ PC ← PC + 4 + BranchAddr	(4) 5 _{hex}
Jump	j J	PC ← JumpAddr	(5) 2 _{hex}
Jump And Link	jal J	$R[31] = PC + 4$; PC ← JumpAddr	(5) 3 _{hex}
Jump Register	jr R	PC ← R[rs]	0 / 08 _{hex}
Load Byte Unsigned	lb I	$R[rt] = \{24'b0, M[R[rs]] + \text{SignExtImm}(7:0)\}$	(2) 24 _{hex}
Load Halfword Unsigned	lhu I	$R[rt] = \{16'b0, M[R[rs]] + \text{SignExtImm}(15:0)\}$	(2) 25 _{hex}
Load Linked	ll I	$R[rt] = M[R[rs]] + \text{SignExtImm}$	(2,7) 30 _{hex}
Load Upper Imm.	lui I	$R[rt] = (\text{imm}, 16'b0)$	0 _{hex}
Load Word	lw I	$R[rt] = M[R[rs]] + \text{SignExtImm}$	(2) 23 _{hex}
Nor	nor R	$R[rd] = \sim (R[rs] R[rt])$	0 / 27 _{hex}
Or	or R	$R[rd] = R[rs] R[rt]$	0 / 25 _{hex}
Or Immediate	ori I	$R[rt] = R[rs] \text{ZeroExtImm}$	(3) 4 _{hex}
Set Less Than	slt R	$R[rd] = (R[rs] < R[rt]) ? 1 : 0$	0 / 2 _{hex}
Set Less Than Imm.	slti I	$R[rt] = (R[rs] < \text{SignExtImm}) ? 1 : 0$	(2) 4 _{hex}
Set Less Than Imm. Unsigned	sltiu I	$R[rt] = (R[rs] < \text{SignExtImm}) ? 1 : 0$	(2,6) 5 _{hex}
Set Less Than Unsig.	sltu R	$R[rd] = (R[rs] < R[rt]) ? 1 : 0$	(6) 0 / 2b _{hex}
Shift Left Logical	sll R	$R[rd] = R[rt] << \text{shamt}$	0 / 00 _{hex}
Shift Right Logical	srl R	$R[rd] = R[rt] >> \text{shamt}$	0 / 02 _{hex}
Store Byte	sb I	$M[R[rs] + \text{SignExtImm}(7:0)] = R[rt](7:0)$	(2) 28 _{hex}
Store Conditional	sc I	$M[R[rs] + \text{SignExtImm}] = R[rt];$ $R[rt] = (\text{atomic}) ? 1 : 0$	(2,7) 38 _{hex}
Store Halfword	sh I	$M[R[rs] + \text{SignExtImm}(15:0)] = R[rt](15:0)$	(2) 29 _{hex}
Store Word	sw I	$M[R[rs] + \text{SignExtImm}] = R[rt]$	(2) 2b _{hex}
Subtract	sub R	$R[rd] = R[rs] - R[rt]$	(1) 0 / 22 _{hex}
Subtract Unsigned	subu R	$R[rd] = R[rs] - R[rt]$	0 / 23 _{hex}

(1) May cause overflow exception

(2) SignExtImm = { 16(immediate[15]), immediate }

(3) ZeroExtImm = { 16(10'b0), immediate }

(4) BranchAddr = { 14(immediate[15]), immediate, 2'b0 }

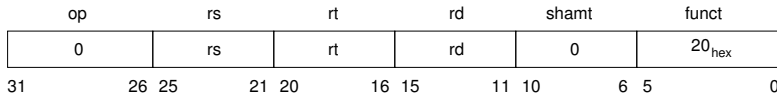
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }

(6) Operands considered unsigned numbers (vs. 2's comp.)

(7) Atomic test/set pair; R[rt] = 1 if pair atomic, 0 if not atomic

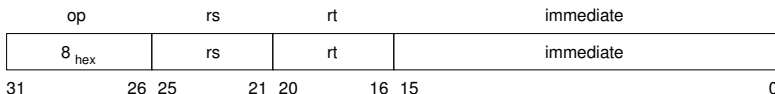
Example: Add

- **Name:** Add
- **Mnemonic:** add
- **Format:** R
- **Operation:** $R[rd] = R[rs] + R[rt]$
- **Note:** (1) May cause overflow exception
- **Opcode/Funct:** $0/20_{\text{hex}}$



Example: Add Immediate

- **Name:** Add Immediate
- **Mnemonic:** addi
- **Format:** I
- **Operation:** $R[rt] = R[rs] + \text{SignExtImm}$
- **Note:** (1) May cause overflow exception
- **Note:** (2) $\text{SignExtImm} = \{16\{\text{immediate}[15]\}, \text{immediate}\}$
- **Opcode/Funct:** 8_{hex}

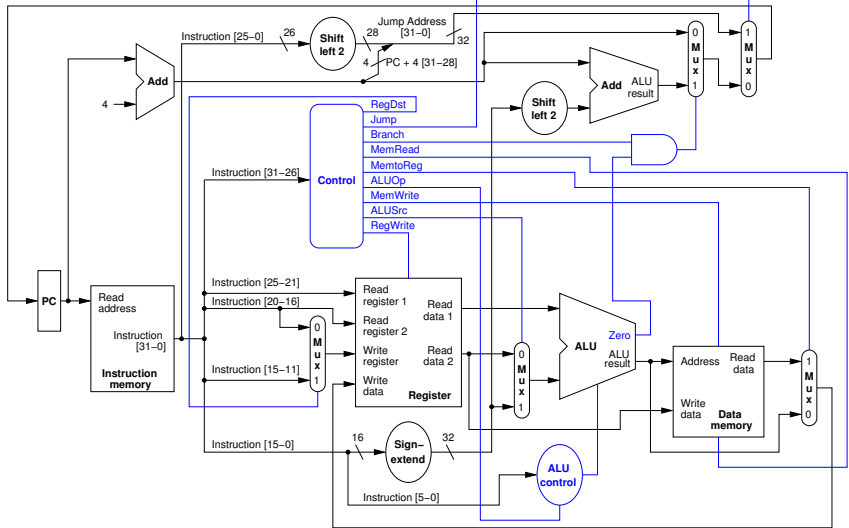


Outline for This Series

Using instructions and their machine languages as a specification:

- ① Arithmetic Logic Unit (ALU)
 - Supports basic arithmetic and logic operations
- ② CPU Datapath
 - Incorporating register file and memories
- ③ Immediate/Address Field
 - How immediate/address field are used
- ④ Jump/Branch Addressing
 - How jump and branch addresses are calculated
- ⑤ Control Unit
 - Control signals of basic instructions

Processor



1. Arithmetic Logic Unit (ALU)

1. Arithmetic Logic Unit (ALU)

Arithmetic Logic Unit (ALU)

Need to support the following operations:

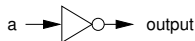
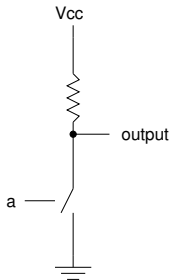
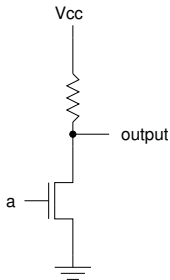
- Arithmetic Operations:
 - Add (add and addi)
 - Subtract (sub)
- Logical Operations:
 - AND (and and andi)
 - OR (or and ori)
 - NOR (nor)
- Comparison Operations:
 - Less than (slt and slti)
 - Equal (beq and bne)
- Memory References
 - Load word (lw)
 - Store word (sw)

NOT Gate

- Desired input/output

Input (a)	Output
0	1
1	0

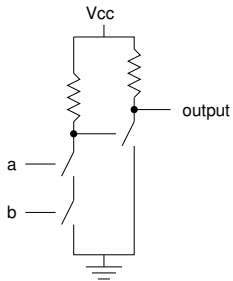
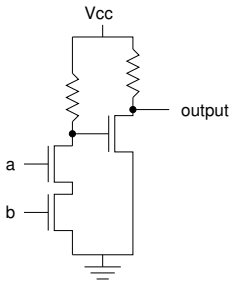
- Can be constructed using MOSFET



AND Gates

- AND Gate

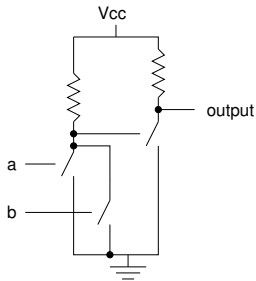
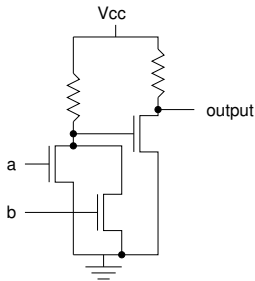
Input (a)	Input (b)	Output
0	0	0
0	1	0
1	0	0
1	1	1



OR Gates

- OR Gate

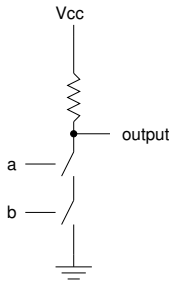
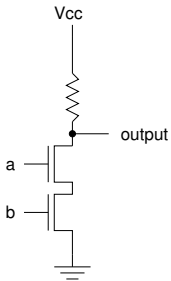
Input (a)	Input (b)	Output
0	0	0
0	1	1
1	0	1
1	1	1



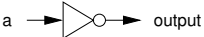


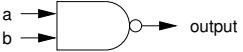
NAND Gate

- NAND Gate

Input (a)	Input (b)	Output
0	0	1
0	1	1
1	0	1
1	1	0



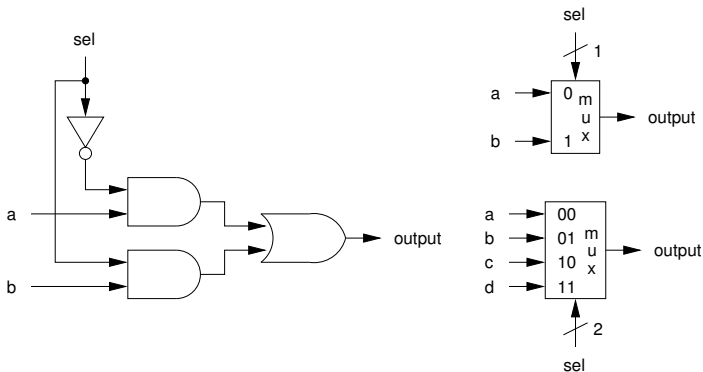
Logic Gates

Name	Symbol	Function
1-input NOT Gate		$\text{output} = !a$
2-input AND Gate		$\text{output} = a \ \&\& \ b$
2-input OR Gate		$\text{output} = a \ \ b$
2-input NAND Gate		$\text{output} = !(a \ \&\& \ b)$

- We know how these gates work
- How to build a CPU using a combination of these gates?

Multiplexer

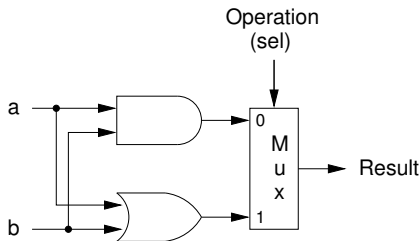
- Desired input/output
 - If sel is 0, output is equal to input a
 - If sel is 1, output is equal to input b
- A multiplexer can be built using AND, OR, and NOT gates



AND and OR Operations (1 Bit)

```
and $t0, $t1, $t2  
or  $t3, $v0, $t4
```

- Use AND gate to produce output for AND operation
- Use OR gate to produce output for OR operation
- Use a multiplexer to select which operation



- a and b are 1-bit inputs

- Recall the instruction add

```
add  $t0, $t1, $t2
```

- We tell CPU to add two numbers (from \$t1 and \$t2)
- Clearly, our CPU must be able to perform addition
- How can we build a circuit that can add two 32-bit numbers?
- From gates that can only perform NOT, AND, OR, NOR, NAND, and XOR?
- Let's think about how we add in both decimal and binary.

One-Bit Adder

- Truth table of the one-bit adder

Input			Output	
A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

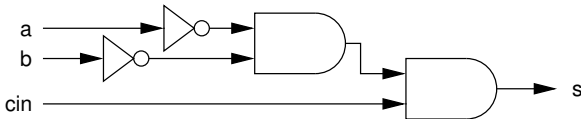
- Outputs that are ones are highlighted
- Example: if $A = 0$, $B = 0$, and $Cin = 1$, we want output S to be 1

Sum of Products

- We want S to be 1 **ONLY** when $A = 0$, $B = 0$, and $C_{in} = 1$.
- Think about an AND gate
 - Output of an AND gate will be 1 if and only if both input a and b are 1s.
- Let's use a 3-input AND gate to produce this output s
 - 3-input AND gate can be built from two 2-input AND gates
- We need all three inputs to be 1 to produce output 1
 - We want output to be 1 when $A = 0$, so invert it using a NOT gate ($\neg A$)
 - We want output to be 1 when $B = 0$, so invert it using a NOT gate ($\neg B$)
 - We want output to be 1 when $C_{in} = 1$, we do not have to do anything
- We obtain output $s = (\neg A \ \&\& \ \neg B) \ \&\& \ C_{in}$

Sum of Products

- Now we know that $s = (!A \ \&\& \ !B) \ \&\& \ \text{Cin}$
- We can build a partial circuit as follows:



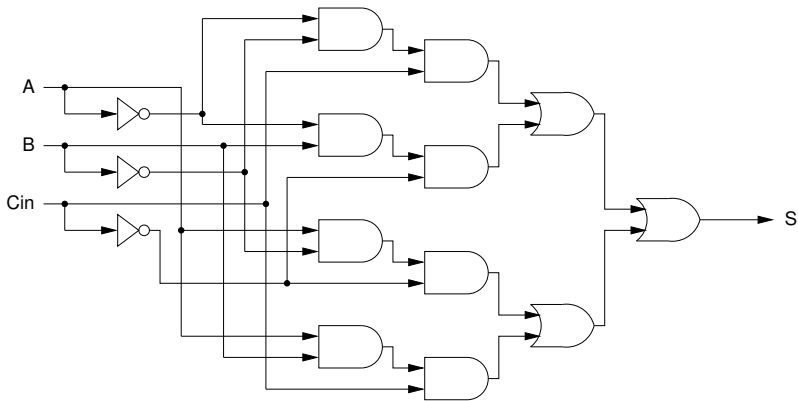
- For simplicity, we use a' to denote $\neg a$ and product to denote AND operator
- Thus, we get $S = A'B'Cin$
- But this is not the only input that produces output $S = 1$
- Output S should be 1 when:
 - $A = 0, B = 0, \text{ and } Cin = 1 \Rightarrow A'B'Cin$
 - $A = 0, B = 1, \text{ and } Cin = 0 \Rightarrow A'BCin'$
 - $A = 1, B = 0, \text{ and } Cin = 0 \Rightarrow AB'Cin'$
 - $A = 1, B = 1, \text{ and } Cin = 1 \Rightarrow ABCin$

Sum of Products

- We need output S to be 1 when one of those set of inputs occurs (from previous slide)
- Recall our OR gate
 - Output of an OR gate will be one if one of its input is one
- Thus we can simply OR them together
- $S = A'B'Cin \parallel A'BCin' \parallel AB'Cin' \parallel ABCin$
- For simplicity, we use $+$ to denote OR operation
 - $S = A'B'Cin + A'BCin' + AB'Cin' + ABCin$
- This is called **Sum of Products**

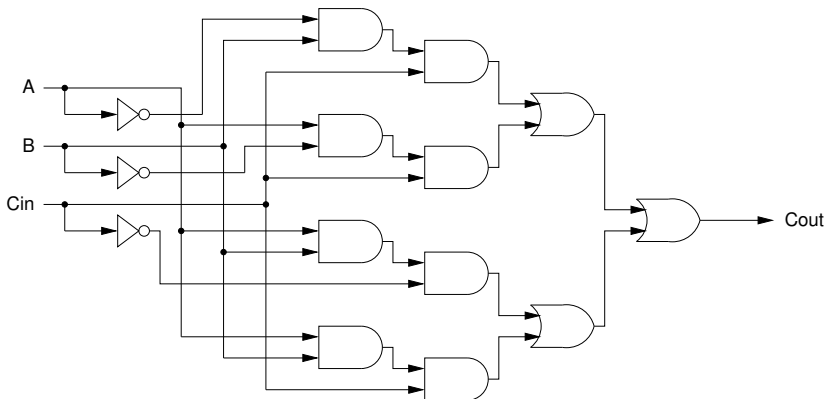
Sum of Products

- Circuit for producing sum (S) of a one-bit adder



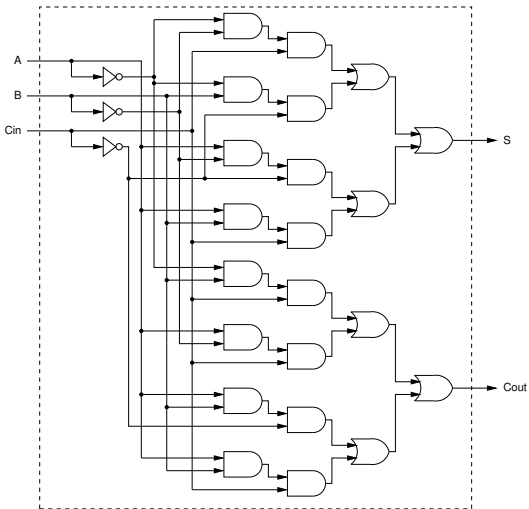
Sum of Products

- Doing the same process for Cout and we get
 - $C_{out} = A'BC_{in} + AB'C_{in} + ABC_{in}' + ABC_{in}$



Sum of Products

- Now, we can build a one-bit adder using a bunch of NOT, AND, and OR gates



Boolean Algebra

- Boolean algebra consists of:
 - Binary Value: 0 and 1
 - Two binary operators: AND (\times or \cdot) and OR ($+$)
 - One unary operator: NOT (\neg)
- Some Properties:
 - Idempotent:
 - $a \cdot a = a$
 - $a + a = a$
 - Commutative
 - $a \cdot b = b \cdot a$
 - $a + b = b + a$
 - Associative
 - $a \cdot (b \cdot c) = (a \cdot b) \cdot c$
 - $a + (b + c) = (a + b) + c$

- Properties:
 - Distributive
 - $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
 - $a + (b \cdot c) = (a + b) \cdot (a + c)$
 - De Morgan's law
 - $\neg(a \cdot b) = \neg a + \neg b$
 - $\neg(a + b) = \neg a \cdot \neg b$
 - Identities
 - $a + (a \cdot b) = a$
 - $a \cdot (a + b) = a$
 - $\neg\neg a = a$
 - $a + \neg a = 1$
 - $a \cdot \neg a = 0$
- With AND, OR, and NOT gates, we can express any function in Boolean algebra

Using NAND Only

- What if we have only NAND gates, can we express all functions in Boolean algebra?
- In other words, can we build AND, OR, and NOT operations, using only NAND gates?
- Note that NAND is NOT AND ($a \text{ nand } b = \neg(a \cdot b)$)
- NOT using NAND

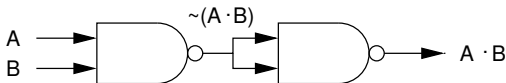
$$\begin{aligned}\neg A &= \neg(A \cdot A) \\ &= A \text{ nand } A\end{aligned}$$



Using NAND Only

- AND using NAND

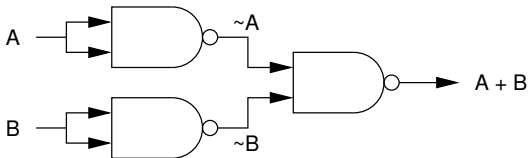
$$\begin{aligned}A \cdot B &= \neg(\neg(A \cdot B)) \\&= \neg(A \text{ nand } B) \\&= \neg((A \text{ nand } B) \cdot (A \text{ nand } B)) \\&= (A \text{ nand } B) \text{ nand } (A \text{ nand } B)\end{aligned}$$



Using NAND Only

- OR using NAND

$$\begin{aligned}A + B &= \neg(\neg(A + B)) \\&= \neg(\neg A \cdot \neg B) \\&= \neg A \text{ nand } \neg B \\&= (\neg(A \cdot A)) \text{ nand } (\neg(B \cdot B)) \\&= (A \text{ nand } A) \text{ nand } (B \text{ nand } B)\end{aligned}$$



Using NOR Only

- Recall that A nor B is $\neg(A + B)$
- NOT using NOR

$$\begin{aligned}\neg A &= \neg(A + A) \\ &= A \text{ nor } A\end{aligned}$$

- AND using NOR

$$\begin{aligned}A \cdot B &= \neg(\neg A) \cdot \neg(\neg B) \\&= \neg((\neg A) + (\neg B)) \\&= (\neg A) \text{ nor } (\neg B) \\&= (\neg(A + A)) \text{ nor } (\neg(B + B)) \\&= (A \text{ nor } A) \text{ nor } (B \text{ nor } B)\end{aligned}$$

- OR using NOR

$$\begin{aligned}A + B &= (A + B) \cdot (A + B) \\&= \neg(\neg((A + B) \cdot (A + B))) \\&= \neg(\neg(A + B) + \neg(A + B)) \\&= \neg(A + B) \text{ nor } \neg(A + B) \\&= (A \text{ nor } B) \text{ nor } (A \text{ nor } B)\end{aligned}$$

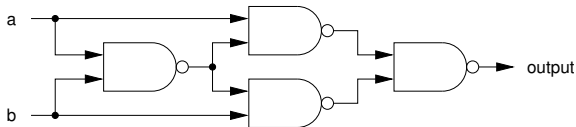
XOR Gate

- XOR Gate

Input (a)	Input (b)	Output
0	0	0
0	1	1
1	0	1
1	1	0



- Output = $a'b + ab'$



XOR Gate

- Output = $a'b + ab'$

$$\begin{aligned}(\neg a \cdot b) + (a \cdot \neg b) &= \neg(\neg((\neg a \cdot b) + (a \cdot \neg b))) \\&= \neg(\neg(\neg a \cdot b) \cdot \neg(a \cdot \neg b)) \\&= \neg(\neg a \cdot b) \text{ nand } \neg(a \cdot \neg b), \text{ but}\end{aligned}$$

$$\begin{aligned}\neg(\neg a \cdot b) &= \neg((\neg a \cdot b) + 0) \\&= \neg((\neg a \cdot b) + (\neg b \cdot b)) \\&= \neg((\neg a + \neg b) \cdot b) \\&= (\neg a + \neg b) \text{ nand } b \\&= (\neg(a \cdot b) \text{ nand } b) \\&= (a \text{ nand } b) \text{ nand } b, \text{ and}\end{aligned}$$

$$\begin{aligned}\neg(a \cdot \neg b) &= \neg(0 + (a \cdot \neg b)) \\&= \neg((a \cdot \neg a) + (a \cdot \neg b)) \\&= \neg(a \cdot (\neg a + \neg b)) \\&= a \text{ nand } (\neg a + \neg b) \\&= a \text{ nand } \neg(a \cdot b) \\&= a \text{ nand } (a \text{ nand } b)\end{aligned}$$

Simplifying Expressions

- Recall our one-bit full-adder
 - $S = A'B'Cin + A'BCin' + AB'Cin' + ABCin$
 - $Cout = A'BCin + AB'Cin + ABCin' + ABCin$
- Look at the last two terms:
 - $ABCin' + ABCin$
- From the boolean algebra
 - $ABCin' + ABCin = AB(Cin' + Cin) = AB$
- Similar to
 - $A'BCin + ABCin = BCin(A' + A) = BCin$
 - $AB'Cin + ABCin = ACin(B' + B) = ACin$
- Thus, the algebraic expression of Cout can be reduced to
 - $Cout = BCin + ACin + AB$
- In doing so, we reduce the complexity of the circuit which will be smaller use less power

Karnaugh Map

- Karnaugh Map is a tool to help simplify boolean expression
- Create a table using **Gray Codes**
 - A binary code where each subsequent value only differs in out bit from the previous code
- For example:

2-bit	3-bit
00	000
01	001
11	011
10	010
	110
	111
	101
	100

- Gray codes can be used for both column and row headings

Karnaugh Map

- Recall that $C_{out} = A'BC_{in} + AB'C_{in} + ABC_{in}' + ABC_{in}$

BC _{in} \ A	0	1
00	0	0
01	0	1
11	1	1
10	0	1

- Fill in the table using C_{out} 's boolean expression
- For example, the highlighted cell corresponds to $AB'C_{in}$

Karnaugh Map

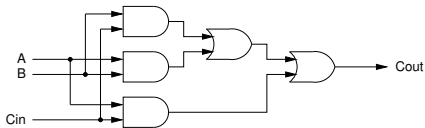
- Looking for adjacent maximum rectangular groups of 1s with power of 2 elements (can be horizontal, vertical, both, or wrap around edges)
- For example:

ACin			BCin			AB		
BCin \ A			BCin \ A			BCin \ A		
0	1		0	1		0	1	
00	0	0	00	0	0	00	0	0
01	0	1	01	0	1	01	0	1
11	1	1	11	1	1	11	1	1
10	0	1	10	0	1	10	0	1

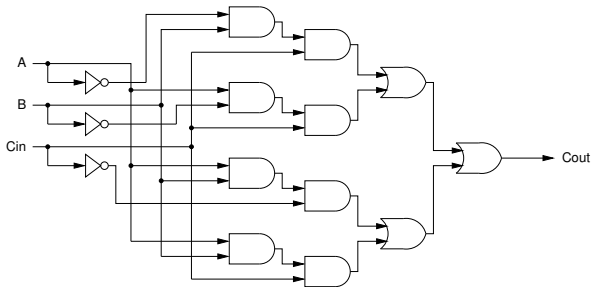
- First table, A and Cin are 1s, B can be either 1 or 0
- Second table, B and Cin are 1s, A can be either 1 or 0
- Thrid table, A and B are 1s, Cin can be either 1 or 0

Karnaugh Map

- New circuit for producing Cout



- Compare to



Karnaugh Map

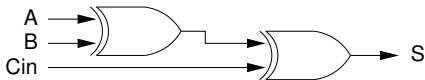
- What about sum (S)?
 - $S = A'B'Cin + A'BCin' + AB'Cin' + ABCin$

		A	
		0	1
Bcin	00	0	1
	01	1	0
	11	0	1
	10	1	0

- We cannot group 1s together. Thus, we cannot simplify the expression of sum.
- Notice that S will be 1 if the number of ones among A, B, and Cin is an odd number (one or three)
- Recall the two-input XOR gate
 - If A is 0 and B is 1, the output is 1
 - If A is 1 and B is 0, the output is 1
 - Otherwise, the output is 0.

Simplify the Expression of Sum

- We can use XOR to produce the output S as follows:



- We can verify using boolean algebra where
 $A \oplus B = A'B + AB'$

$$\begin{aligned}(A \oplus B) \oplus Cin &= (A'B + AB') \oplus Cin \\&= (A'B + AB')' Cin + (A'B + AB') Cin' \\&= ((A'B)'(AB')') Cin + (A'B + AB') Cin' \\&= ((A'' + B')(A' + B'')) Cin + (A'B + AB') Cin' \\&= ((A + B')(A' + B)) Cin + (A'B + AB') Cin' \\&= ((AA' + A'B') + (AB + BB')) Cin + (A'B + AB') Cin' \\&= (A'B' + AB) Cin + (A'B + AB') Cin' \\&= A'B' Cin + ABCin + A'BCin' + AB' Cin'\end{aligned}$$

Karnaugh Map (Four Variables)

- We can use Karnaugh Map with more than three variables.
- For example, four variables:

AB \ CD	CD			
	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	1	1	0
10	0	1	1	0

- From the above table, the output is 1 when D and A are ones.
- Thus, we have $\text{Output} = AD$

Karnaugh Map (Four Variables)

- For example, four variables:

AB \ CD	CD			
	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	1	1	1	1
10	0	0	0	0

- From the above table, the output is 1 when B is one.
- Thus, we have $\text{Output} = B$

Karnaugh Map (Four Variables)

- For example, four variables:

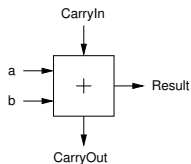
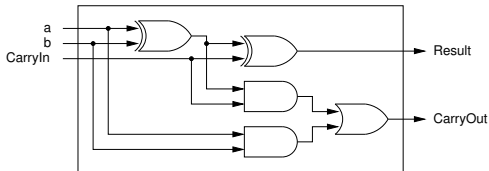
AB \ CD	CD			
	00	01	11	10
00	1	0	0	1
01	0	0	0	0
11	0	0	0	0
10	1	0	0	1

- From the above table, the output is 1 when B and D are zeros.
- Thus, we have $\text{Output} = B'D'$

Addition (1 Bit)

```
add $t0, $t1, $t2
```

- Two inputs for the operands (a and b)
- One input for taking carry from previous adder (CarryIn)
- One output for sum (Result)
- One output to pass the carry on to the next adder (CarryOut)
- Can be constructed as follows:



Addition (1 Bit)

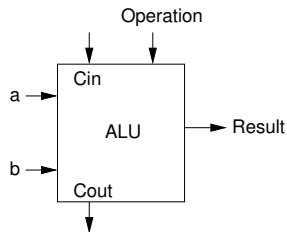
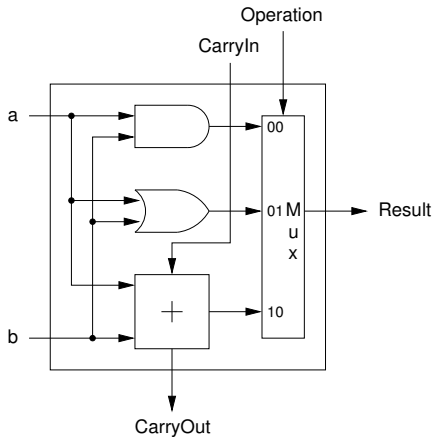
- From the circuit in previous slide:
 - Sum (Result) = $(A \oplus B) \oplus \text{Cin}$ — We already discussed this
 - Cout (CarryOut) = $(A \oplus B)\text{Cin} + AB$???
- From the boolean algebra:

$$\begin{aligned}(A \oplus B)\text{Cin} + AB &= (A'B + AB')\text{Cin} + AB \\&= A'BCin + AB' Cin + AB \\&= A'BCin + AB' Cin + AB(\text{Cin} + \text{Cin}') \\&= A'BCin + AB' Cin + ABCin + ABCin'\end{aligned}$$

Which is the same as out Cout analysed from the truth table

Addition with AND and OR

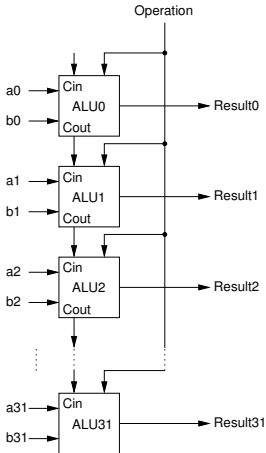
- Need to incorporate with AND and OR operations
- Need to expand multiplexer to select among AND, OR, and addition operations



- Can be considered as 1-bit ALU

32-Bit ALU

- Construct 32-bit-wide ALU using a chain of 32 1-bit-wide ALU
- Let x_i mean the i th bit of x



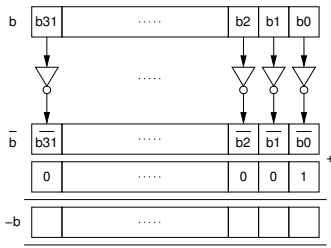
- How about operand(s) with negative value in two's complement (e.g., $6 + (-5)$, $(-1) + 3$, or $(-2) + (-3)$)

Subtraction Operation

```
sub $t5, $a0, $a1
```

- $x - y = x + (-y)$
- We can use existing addition operation to perform subtraction
- Need a logic to negate an operand
- MIPS uses two's complement for signed number representation
- How to negate a two's complement number?

$$-x = \bar{x} + 1$$



Subtraction Examples (4-bit representation)

- Example: $5 - 4$

$$\begin{array}{r} 0101 \\ 1100 \quad + \\ \hline 0001 \\ \hline \hline \end{array}$$

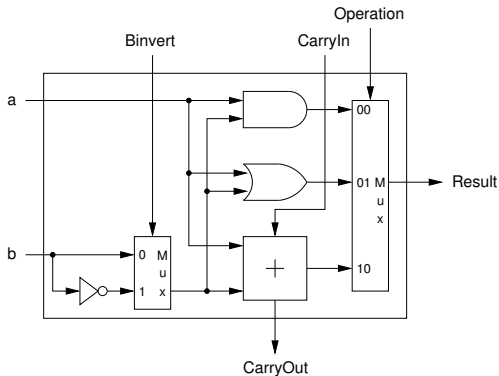
- Example: $-5 + 4$

$$\begin{array}{r} 1011 \\ 0100 \quad + \\ \hline 1111 \\ \hline \hline \end{array}$$

- Example: $-1 - 2$

$$\begin{array}{r} 1111 \\ 1110 \quad + \\ \hline 1101 \\ \hline \hline \end{array}$$

ALU with Subtraction



- For ALU0 (least significant bit ALU)
 - *CarryIn* must be 1 for subtraction
 - *CarryIn* must be 0 for addition

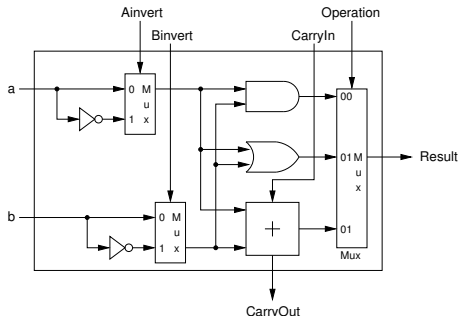
NOR Operation

```
nor $t2, $t1, $a2
```

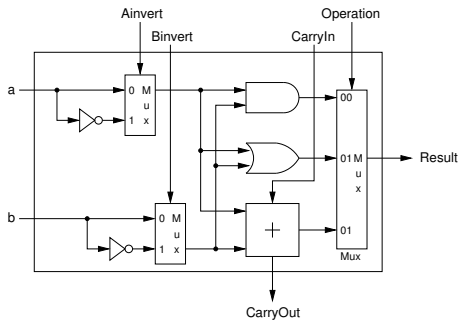
- Recall the definition of nor:

$$a \text{ NOR } b = \neg(a \vee b) = \neg a \wedge \neg b$$

- We already have inverse of b and the AND operation
- We need an inverter in every ALU to invert a input
- Need a multiplexer to select between a or $\neg a$



ALU Operations



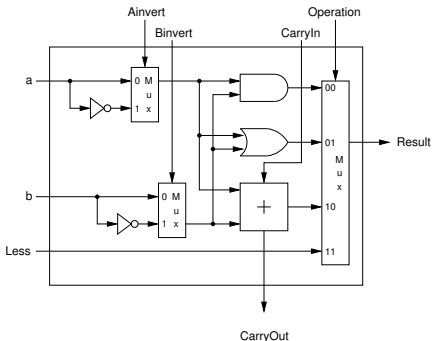
Operation	Ainvert	Binvert	CarryIn (first ALU)	Operation
AND	0	0	N/A	00 ₂
OR	0	0	N/A	01 ₂
Addition	0	0	0	10 ₂
Subtraction	0	1	1	10 ₂
NOR	1	1	N/A	00 ₂

- Binvert and CarryIn of the first ALU are always the same
- Put them together and call it Bnegate

Set on Less Than Operation

```
slt $t0, $s0, $s1
```

- Sets register \$t0 to 1 if $\$s0 < \$s1$. Otherwise set \$t0 to 0.
- Set all but the least significant bit of the ALU to 0
- Need to expand the output multiplexer to select the new input called Less
- Less is always 0 for ALU1 to ALU31



Set on Less Than Operation

- How to compare whether a is less than b ?

$$a < b \equiv (a - b) < (b - b) \equiv (a - b) < 0$$

- If $(a - b) < 0$, then $a < b$
- We can use subtract to compare a and b .
- How do we know that the result of $a - b$ is less than 0?
- If sign bit of a two's complement number is 1, the number is a negative number
- Examples (4-bit representation)
 - Check whether 2_{10} is less than 5_{10}

$$2_{10} - 5_{10} = 2_{10} + (-5_{10}) = 0010_2 + 1011_2 = 1101_2$$

The sign bit is 1. 2_{10} is less than 5_{10} .

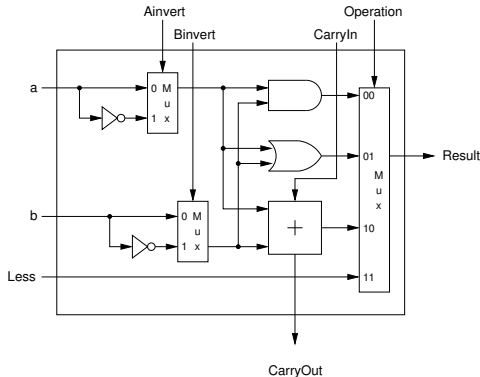
- Check whether -3_{10} is less than -1_{10}

$$-3_{10} - (-1_{10}) = -3_{10} + 1_{10} = 1101_2 + 0001_2 = 1110_2$$

The sign bit is 1. -3_{10} is less than -1_{10}

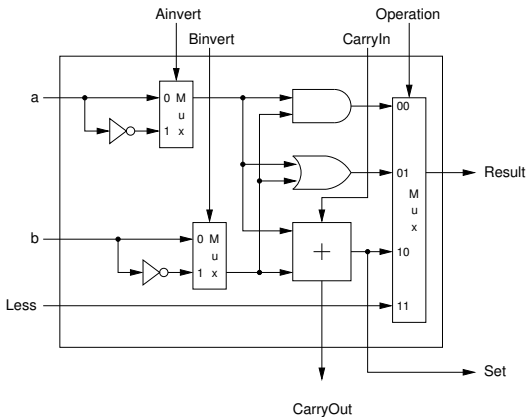
Set on Less Than Operation

- We need to know the sign bit of the result of subtraction operation
- We cannot simply use the output of ALU31



- The output of the ALU31 is 0 for set on less than operation

Set on Less Than Operation



- Need to extract the output of the **adder** of ALU31 called Set signal
- Connect Set signal to the Less input of the least significant bit ALU (ALU0)

Problem with Set on Less Than

- Unfortunately, an overflow can occur
- Consider the following 4-bit representation examples:
 - Check whether -7_{10} is less than 6_{10}

$$-7_{10} - 6_{10} = -7_{10} + (-6_{10}) = 1001_2 + 1010_2 = 0011_2$$

The sign bit is 0. Thus -7_{10} **is not less than** 6_{10}

- Check whether 7_{10} is less than -1_{10}

$$7_{10} - (-1_{10}) = 7_{10} + 1_{10} = 0111_2 + 0001_2 = 1000_2$$

The sign bit is 1. Thus 7_{10} **is less than** -1_{10}

- Check whether 6_{10} is less than -3_{10}

$$6_{10} - (-3_{10}) = 6_{10} + 3_{10} = 0110_2 + 0011_2 = 1001_2$$

The sign bit is 1. Thus 6_{10} **is less than** -3_{10}

- Need to check whether an overflow occurs

- In addition, no overflow if two operands have different signs
- In addition, overflow occurs in the following situations:
 - Adding two positive numbers results in a negative number

$$7_{10} + 1_{10} = 0111_2 + 0001_2 = 1000_2 = -8_{10}$$

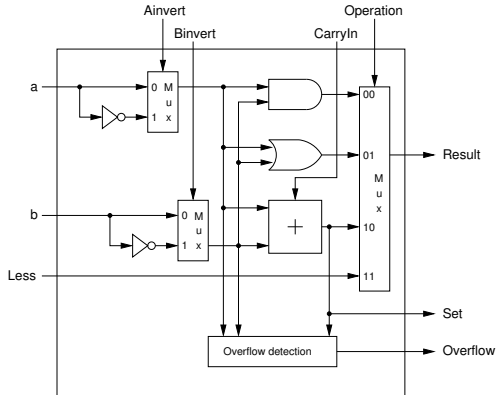
- Adding two negative numbers results in a positive number

$$-8_{10} + (-1_{10}) = 1000_2 + 1111_2 = 0111_2 = 7_{10}$$

- In subtraction, no overflow if two operands have the same signs
- In subtraction, overflow occurs in the following situations:
 - Subtracting a negative number from a positive number results in a negative number
 - Subtracting a positive number from a negative number results in a positive number

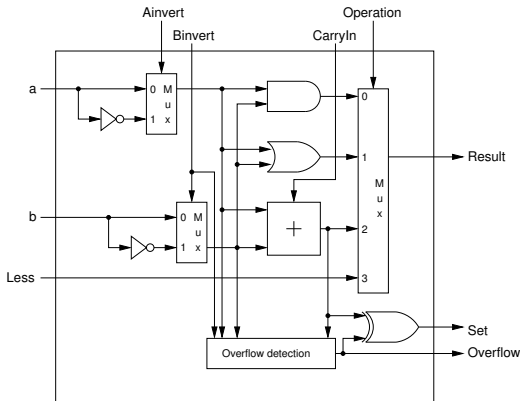
Overflow

- To detect overflow, we need the following:
 - The sign bit of the first operand
 - The sign bit of the second operand
 - The sign bit of the result

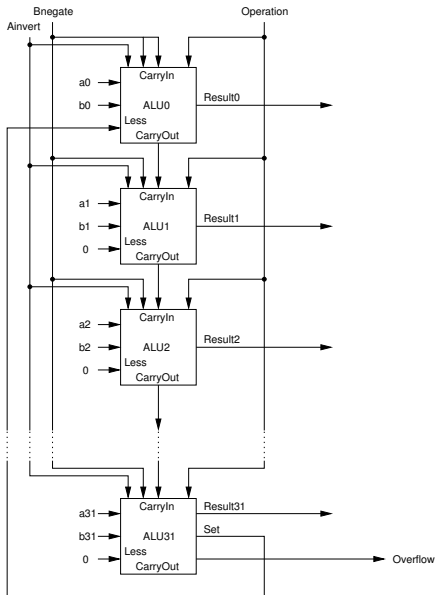


Fixing the Set Output

- We need to invert the original Set output if the overflow occur ($Overflow = 1$)
- Can be easily done by adding an XOR gate



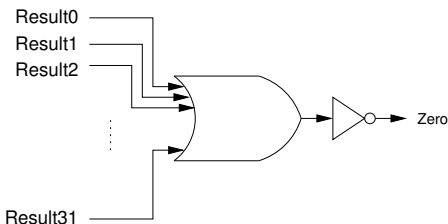
ALU with Set on Less Than



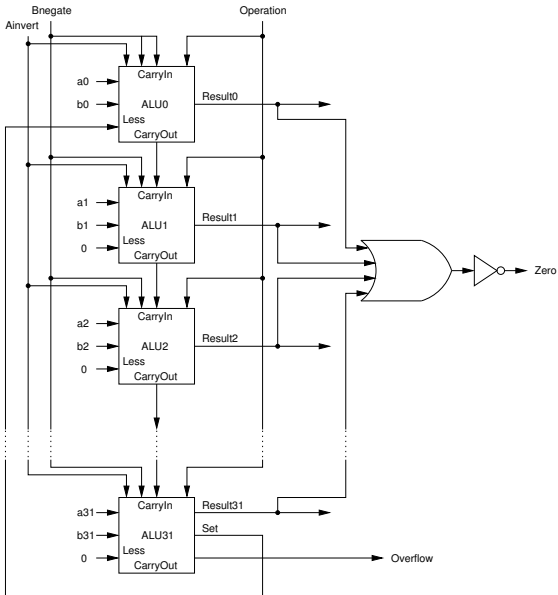
Branches (beq and bne)

```
beq $t0, $t1, L1  
bne $s0, $s1, L2
```

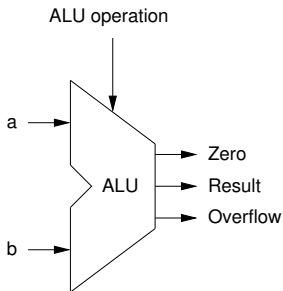
- Need the ability to check whether a is equal to b
- If $a = b$, $(a - b) = 0$
- Use subtraction and check whether the result is equal to 0
- Need a new output called Zero
- Zero will be 1 if $a = b$. Otherwise, 0.
- Simply OR all ALU outputs and inverse the result



ALU



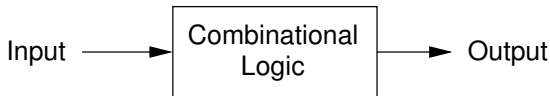
ALU



Operation	ALU operation		
	Ainvert	Bnegate	Operation
AND	0	0	00 ₂
OR	0	0	01 ₂
Addition	0	0	10 ₂
Subtraction	0	1	10 ₂
NOR	1	1	00 ₂
Set on Less Than	0	1	11 ₂
Branch	0	1	N/A

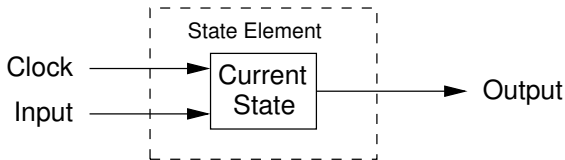
Combinational Logic

- An Output of a combinational logic depends solely on its inputs
- Given a set of inputs, it always produces the same output.
- Examples: gates, adder, and ALU



Sequential Logic

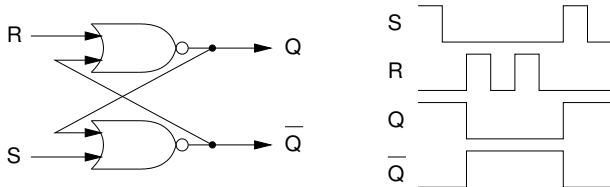
- An output of a sequential logic depends on its inputs and **the content of its internal state**
- Contains at least one **state element** to store its state
 - A state element has at least two inputs
 - **Data** to be written to the element, and
 - **Clock** to determines when the data value is written
- Examples: registers and memory



State Elements

- A state element need to hold a value
- When its output is changed, the output stays the same
 - If the output was 1, maintains output 1 until a specific input is detected
 - If the output was 0, maintains output 0 until a specific input is detected
- This type of circuit can be built using gates with feedback(s)
- With feedback(s), we need to assume the initial state
 - Initial state is its output when the power is just turn on

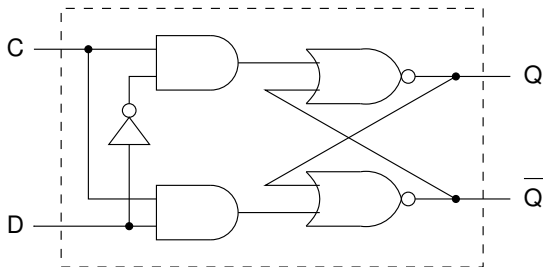
S-R Latch



S	R	Q	\bar{Q}	Action
0	0	Q	\bar{Q}	Hold state (values)
1	0	1	0	Set
0	1	0	1	Reset
1	1	X	X	Not allowed

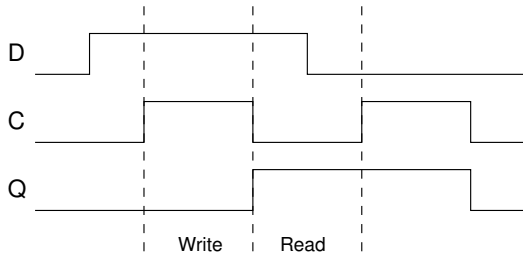
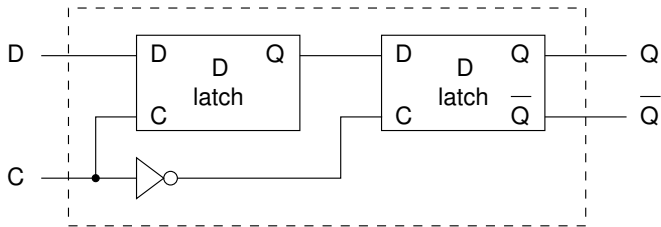
- \bar{Q} is the complement (negate/invert) of Q
- Assume that its initial state is $Q = 0$ and $\bar{Q} = 1$
- The value of Q and \bar{Q} when S and R are zeros depend on previous values of S and R.

D Latch

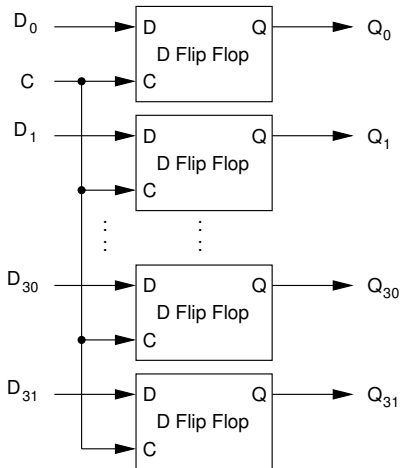


- Constructed from S-R Latch
 - Recall that both S and R cannot be 1 at the same time
 - Can outputs of two AND gates be 1 at the same time?
- D Latch
 - When C is 1 (asserted), the latch is open. Output Q becomes the value of input D.
 - When C is 0 (deasserted), the latch is closed. Output Q is whatever value was stored the last time the latch is open.

D Flip Flop

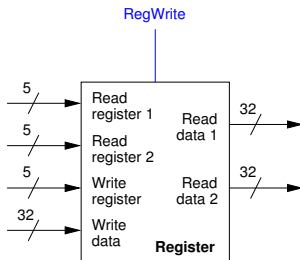


A 32-Bit Register



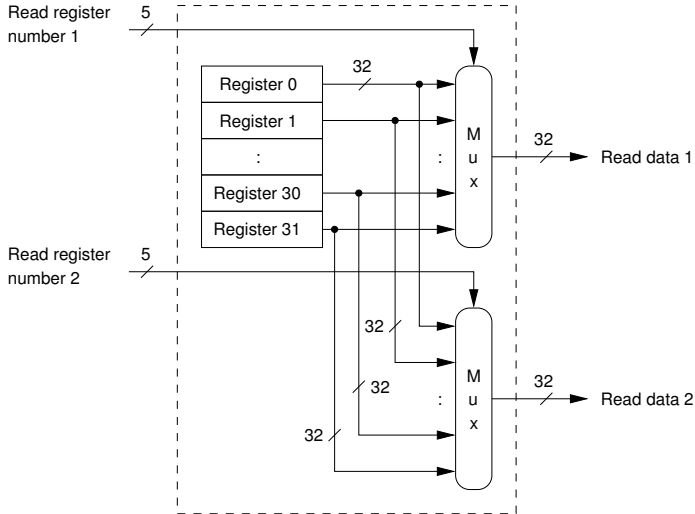
- Use one D flip flop to store 1 bit
- We need 32 D flip flops

Register File



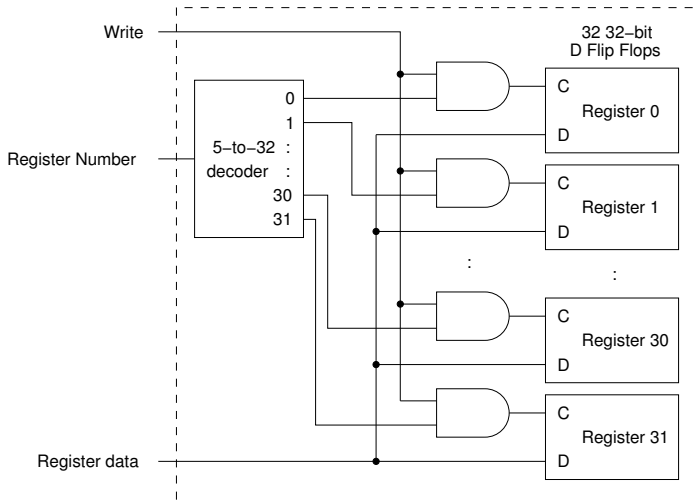
- Our MIPS has 32 general purpose registers, we need 5 bits to indentify which register
- R-type instructions such as add, sub, and and, require two source registers and one destination register
 - The first source operand goes to Read register 1 and its output is Read data 1
 - The second source operand goes to Read register 2 and its output is Read data 2
 - The destination register goes to Write register and the value to be written goes to Write data

Output of A Register File



- Each multiplexer has 32 32-bit inputs and one 32-bit output

Write Port of A Register File



- A circuit that behaves like a function can be built
 - Using logic gates
 - Truth table (sum of products)
 - Simplify using boolean algebra or Karnaugh Map
- Our ALU supports add, sub, and, or, nor, slt, beq, and bne
- A register/memory for storing data can be built
 - Using logic gates with feedback(s)
 - Use clock to determine when to read/write a state element