Procedure?  

Assume caller puts $a0=$a1 = a.b.c.d and wants result in $v0  

```
add $v0,$a0,$a1  // store return value in $v0
lw $a0,1(sp)    // restore registers
lw $a0,4(sp)    // (optional if MIPS convention)
add $v0,$a0,$v0  // 'pop' the stack
jr $ra         // The actual return to calling routine
```  

Example: Leaf _procedure()  

```
Example: Leaf_procedure()  

• Procedure?  

Assume caller puts $a0=$a1 = a.b.c.d and wants result in $v0  

```
add $v0,$a0,$a1  // store return value in $v0
lw $a0,1(sp)    // restore registers
lw $a0,4(sp)    // (optional if MIPS convention)
add $v0,$a0,$v0  // 'pop' the stack
jr $ra         // The actual return to calling routine
```  

Example: Swap()  

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

Example: Nest ed _procedure()  

```
Example: Nested_procedure()  

• What about nested procedures? $a0 ??  

• Recursive procedures?  

Assume $a0 = n  

```
fact sub $gp,$a0, 8  // Make space for 2 temp locations
lw $a0,4(sp) // save return address
lw $a0,1(sp) // save argument  
add $v0,$a0,$a0  // store return value in $v0
lw $v0,1(sp)    // restore registers
lw $v0,4(sp)    // (optional if MIPS convention)
add $v0,$a0,$v0  // 'pop' the stack
jr $ra         // The actual return to calling routine
```  

Example: Nested_procedure()  

```
Example: Nested_procedure()  

• What about nested procedures? $a0 ??  

• Recursive procedures?  

Assume $a0 = n  

```
fact sub $gp,$a0, 8  // Make space for 2 temp locations
lw $a0,4(sp) // save return address
lw $a0,1(sp) // save argument  
add $v0,$a0,$a0  // store return value in $v0
lw $v0,1(sp)    // restore registers
lw $v0,4(sp)    // (optional if MIPS convention)
add $v0,$a0,$v0  // 'pop' the stack
jr $ra         // The actual return to calling routine
```  

Example: Nest ed _procedure()  

```
Example: Nested_procedure()  

• What about nested procedures? $a0 ??  

• Recursive procedures?  

Assume $a0 = n  

```
fact sub $gp,$a0, 8  // Make space for 2 temp locations
lw $a0,4(sp) // save return address
lw $a0,1(sp) // save argument  
add $v0,$a0,$a0  // store return value in $v0
lw $v0,1(sp)    // restore registers
lw $v0,4(sp)    // (optional if MIPS convention)
add $v0,$a0,$v0  // 'pop' the stack
jr $ra         // The actual return to calling routine
```  

Example: Nested_procedure()  

```
Example: Nested_procedure()  

• What about nested procedures? $a0 ??  

• Recursive procedures?  

Assume $a0 = n  

```
fact sub $gp,$a0, 8  // Make space for 2 temp locations
lw $a0,4(sp) // save return address
lw $a0,1(sp) // save argument  
add $v0,$a0,$a0  // store return value in $v0
lw $v0,1(sp)    // restore registers
lw $v0,4(sp)    // (optional if MIPS convention)
add $v0,$a0,$v0  // 'pop' the stack
jr $ra         // The actual return to calling routine
```  

Example: Swap()  

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

Example: Swap()  

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

Example: Swap()  

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

Example: Swap()  

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

Example: Swap()  

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

Example: Swap()  

```
swap(int v[]), int k) ;  
{  int temp = v[k];  
v[k] = *(v+k+1);  
*(v+k) = temp;  
}
```
Comparing Instruction Set Architectures

Design-time metrics:
- Can it be implemented, how long, at what cost?
- Can it be programmed? Ease of compilation?

Static Metrics:
- How many bytes does the program occupy in memory?
- How many instructions does the processor fetch to execute the program?
- How many clocks are required per instruction?
- How "lean" an clock is practical?

Best Metric: Time to execute the program!

This depends on:
- instruction set,
- processor organization, and
- compilation techniques.

Computer Performance

Measuring and Discussing Computer System Performance

or
“My computer is faster than your computer”

SPEC Performance

Performance depends on the eyes of the beholder?

- Purchasing perspective
  - given a collection of machines, which has the
    • best performance?
    • least cost?
    • best performance / cost?
- Design perspective
  - faced with design options, which has the
    • best performance improvement?
    • least cost?
    • best performance / cost?
- Both require
  - basis for comparison
  - metric for evaluation
- Our goal is to understand cost & performance implications of architectural choices

Two ideas

Two mechanisms of getting to the bay-area

<table>
<thead>
<tr>
<th>Plane</th>
<th>DC to Paris</th>
<th>Speed</th>
<th>Passengers</th>
<th>Throughput (rwwn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boeing 747</td>
<td>6.5 hours</td>
<td>610 mph</td>
<td>470</td>
<td>286,700</td>
</tr>
<tr>
<td>Concorde</td>
<td>3 hours</td>
<td>1350 mph</td>
<td>132</td>
<td>178,200</td>
</tr>
</tbody>
</table>

• How much faster is the Concorde compared to the 747?
• How much bigger is the 747 than the Douglas DC-8?

Which has higher performance?

- Time to do the task (Execution Time)
  - execution time, response time, latency
- Tasks per day, hour, week, sec, ns ... (Performance)
  - throughput, bandwidth

Response time and throughput often are in opposition

<table>
<thead>
<tr>
<th>Vehicle</th>
<th>Time to Bay Area</th>
<th>Speed</th>
<th>Passengers</th>
<th>Throughput (gpm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Freight</td>
<td>3.1 hours</td>
<td>180 mph</td>
<td>2</td>
<td>320</td>
</tr>
<tr>
<td>Greyhound</td>
<td>7.7 hours</td>
<td>65 mph</td>
<td>60</td>
<td>39/30</td>
</tr>
</tbody>
</table>

• Time to do the task from start to finish
  - execution time, response time, latency
• Tasks per unit time
  - throughput, bandwidth
  - mostly used for data movement

Response time and throughput often are in opposition
Relative performance?

- can be confusing
  - A runs in 12 seconds
  - B runs in 20 seconds
  - A/B = 6, so A is 40% faster, or 1.6X faster, or B is 67% slower
- needs a precise definition

How many times?

- Time of Concorde vs. Boeing 747?
  - Concorde is 1350 mph / 610 mph = 2.2 times faster
  - 6.5 hours / 3 hours
- Throughput of Concorde vs. Boeing 747?
  - Concorde is 286,700 pmpm / 268,000 pmpm = 1.6 times faster
- Boeing is 1.6 times (60%) faster in terms of throughput
- Concorde is 2.2 times (120%) faster in terms of flying time

We will focus primarily on execution time for a single job.

Relative performance?

- Performance is in units of things-per-second
  - bigger is better
- If we are primarily concerned with response time
  - performance(x) = execution_time(x)

\[ \text{Relative Performance} = \frac{\text{Performance}_X}{\text{Performance}_Y} \]

Some grammar?

- "times faster than" (or "times as fast as")
  - there's a multiplicative factor relating quantities
  - "X was 3 time faster than Y" = speed(X) = 3 speed(Y)
- "percent faster than"
  - implies an additive relationship
  - "X was 25% faster than Y" = speed(X) = (1+25/100) speed(Y)
- "percent slower than"
  - implies subtraction
  - "X was 5% slower than Y" = speed(X) = (1-5/100) speed(Y)
  - "100% slower" means it doesn't move at all!
- "times slower than" or "times as slow as"
  - is awkward.
  - "X was 3 times slower than Y" means speed(X) = 1/3 speed(Y)

Avoid Linguistic Confusion

\[ X \text{ is } r \text{ times faster than } Y \rightarrow \text{ speed}(X) = r \text{ speed}(Y) \]
\[ \rightarrow \text{ speed}(Y) = \frac{1}{r} \text{ speed}(X) \]
\[ \rightarrow Y \text{ is } r \text{ times slower than } X \]

\[ X \text{ is } r \text{ times faster than } Y, \& Y \text{ is } s \text{ times faster than } Z \]
\[ \rightarrow \text{ speed}(X) = r \text{ speed}(Y) = rs \text{ speed}(Z) \]
\[ \rightarrow X \text{ is } rs \text{ faster than } Z \]

(Cannot do this with % numbers!)

Easiest way to avoid confusion:
- Convert "% faster" to "times faster"
- then do calculation and convert back if needed.
- Example: change "25% faster" to "5/4 times faster".

Which time anyways?

- user CPU time?
- total CPU time (user + kernel)? (includes op. sys. code)
- Wallclock time? (total elapsed time)
  - Includes time spent waiting for I/O, other users, ...
- Answer depends ...
  - For measuring processor speed, we can use total CPU.
  - If no I/O or interrupts, wallclock may be better
  - more precise (microseconds rather than 1/100 sec)
  - can measure individual sections of code
Metrics of Performance

![Diagram](image)

Each metric has a place and a purpose, and each can be misused.

Levels of benchmarking

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>• representative</td>
</tr>
<tr>
<td>• portable</td>
<td></td>
</tr>
<tr>
<td>• widely used</td>
<td></td>
</tr>
<tr>
<td>• improvements useful in reality</td>
<td>• less representative</td>
</tr>
</tbody>
</table>

Cycle Time

- Instead of reporting execution time in seconds, we often use cycles
- Clock "ticks" indicate when to start activities (one abstraction)
- cycle time = time between ticks = seconds/cycle
- clock rate (frequency) = cycles per second (1 Hz = 1 cycle/sec)
- A 200 MHz clock has a cycle time of:

\[
\frac{1}{200 \times 10^9} = 5 \text{ nanoseconds}
\]

Performance Variation

<table>
<thead>
<tr>
<th>Program Combination</th>
<th>Number of Instructions</th>
<th>CPI</th>
<th>Clock Cycle Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same machine different programs</td>
<td>different</td>
<td>similar</td>
<td>same</td>
</tr>
<tr>
<td>Same programs, different machines, same ISA</td>
<td>same</td>
<td>different</td>
<td>different</td>
</tr>
<tr>
<td>Same programs, different machines</td>
<td>somewhat different</td>
<td>different</td>
<td>different</td>
</tr>
</tbody>
</table>

Amdahl's Law

\[
\text{Execution Time After Improvement} = \frac{\text{Execution Time Unaffected}}{1 + \text{Amount of Improvement}}
\]

- Example:
  - "Suppose a program runs in 100 seconds on a machine, with multiply responsible for 60 seconds of this time. How much do we have to improve the speed of multiplication if we want the program to run 4 times faster?"
  - How about making it 5 times faster?
  - Principle: Make the common case fast
MIPS, MFLOPS etc.

- **MIPS** - million instructions per second
  - number of instructions executed in program / Clock rate
    - execution time in seconds * 10^6
  - MFLOPS - million floating point operations per second
    - number of floating point operations executed in program
      - execution time in seconds * 10^6

- program-independent
- deceptive

Example RISC Processor

<table>
<thead>
<tr>
<th>Base Machine (Reg / Reg)</th>
<th>Op</th>
<th>Freq</th>
<th>Cycles</th>
<th>CPI(i)</th>
<th>% Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>50%</td>
<td>1</td>
<td>.5</td>
<td>23%</td>
<td></td>
</tr>
<tr>
<td>Load</td>
<td>20%</td>
<td>5</td>
<td>1.0</td>
<td>45%</td>
<td></td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
<td>3</td>
<td>.3</td>
<td>14%</td>
<td></td>
</tr>
<tr>
<td>Branch</td>
<td>20%</td>
<td>2</td>
<td>.4</td>
<td>18%</td>
<td></td>
</tr>
</tbody>
</table>

Typical Mn

How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?

How does this compare with using branch prediction to shave a cycle off the branch time?

What if two ALU instructions could be executed at once?

SPEC

**Which Programs?**

- peak throughput measures (simple programs)
- synthetic benchmarks (whetstone, dhrystone,...)
- Real applications
- SPEC (best of both worlds, but with problems of their own)
  - System Performance Evaluation Cooperative
    - Provides a common set of real applications along with strict guidelines for how to run them.
    - provides a relatively unbiased means to compare machines.

SPEC89

**Compiler "enhancements" and performance**

SPECCPU2000 Suite

- SPECint2000
  - gzip and bzip2 - compression
  - gcc - compiler; 200K lines of messy code!
  - crafty - chess program
  - parser - word processing
  - vortex - object-oriented database
  - perlbench - PERL interpreter
  - eon - computer visualization
  - vpr, twolf - CAD tools for VLSI
  - mcf, gap - "combinatorial" programs
- SPECfp2000 - 10 Fortran, 3 C programs
  - scientific application programs (physics, chemistry, image processing, number theory, ...)

Performance is always misleading

- Performance is specific to a particular program
  - Total execution time is a consistent summary of performance

- For a given architecture performance increases come from:
  - increases in clock rate (without adverse CPI affects)
  - improvements in processor organization that lower CPI
    - compiler enhancements that lower CPI and/or instruction count

- Pitfall: expecting improvement in one aspect of a machine's performance to affect the total performance

- You should not always believe everything you read! Read carefully!
Computer Arithmetic

What do all those bits mean now?

- Instruction
- 1-format
- 2-format
- R-format
- Integer
- Signed
- Floating point
- Unsigned
- Single precision
- Double precision

Computer Arithmetic

- How do you represent
  - Negative numbers?
  - Fractions?
  - Really large numbers?
  - Really small numbers?
- How do you
  - Do arithmetic?
  - Identify errors (e.g., overflow)?
- What is an ALU and what does it look like?
  - ALU: Arithmetic Logic Unit

Big Endian vs. Little Endian

Some processors (e.g., PowerPC) provide both
- If you can figure out how to switch modes or get the compiler to issue “Byte-reversed load’s and store’s”

Binary Numbers: An Introduction

Consider a 4-bit binary number

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>7</td>
</tr>
</tbody>
</table>

Examples of binary arithmetic:

- \(3 + 2 = 5\)
- \(3 + 3 = 6\)

<table>
<thead>
<tr>
<th>3 + 2 = 5</th>
<th>3 + 3 = 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dec</td>
<td>Binary</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>0 1 1 0</td>
</tr>
</tbody>
</table>

Negative Numbers: Some options

- Sign Magnitude – MSB is sign bit, rest the same
  - \(-1 = 1001\)
  - \(-5 = 1101\)
- One’s complement – Flip all bits to negate
  - \(-1 = 1111\)
  - \(-5 = 1010\)

- We would like a number system that provides
  - Obvious representation of 0, 1, 2...
  - Uses adder for addition
  - Single value of 0
  - Equal coverage of positive and negative numbers
  - Easy detection of sign
  - Easy negation

Negative Numbers: Two’s Complement

- Positive numbers: normal binary representation
- Negative numbers: flip bits (0 \(\rightarrow\) 1), then add 1

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Two’s Complement Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8</td>
<td>1000</td>
</tr>
<tr>
<td>-7</td>
<td>1001</td>
</tr>
<tr>
<td>-6</td>
<td>1010</td>
</tr>
<tr>
<td>-5</td>
<td>1011</td>
</tr>
<tr>
<td>-4</td>
<td>1100</td>
</tr>
<tr>
<td>-3</td>
<td>1101</td>
</tr>
<tr>
<td>-2</td>
<td>1110</td>
</tr>
<tr>
<td>-1</td>
<td>1111</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
</tr>
</tbody>
</table>

Smallest 4-bit number: \(-8\)
Biggest 4-bit number: 7
Two’s complement arithmetic

<table>
<thead>
<tr>
<th>Decimal</th>
<th>2’s Complement Binary</th>
<th>Decimal</th>
<th>2’s Complement Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>-1</td>
<td>1111</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>-2</td>
<td>1110</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>-3</td>
<td>1101</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>-4</td>
<td>1100</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>-5</td>
<td>1011</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>-6</td>
<td>1010</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>-7</td>
<td>1001</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>-8</td>
<td>1000</td>
</tr>
</tbody>
</table>

Examples: 7 - 6 = 7 + (-6) = 13 - 5 = 3 + (-5) = -2

Uses simple adder for + and - numbers

Overflow detection again

```
1 0 1 0 1 5
+ 0 1 1 0 -4
--- 1 0 1 1 1
```

So how do we detect overflow?

Carry into MSB = Carry out of MSB

Things to keep in mind

- **Negation**
  - Flip bits and add 1. (Works for + and -)
  - Might cause overflow

- **Extend sign when loading into large register**
  - +3 => 0011, 00000011, 0000000000000001
  - -3 => 1101, 11111101, 1111111111111111

- **Overflow detection**
  - (need to raise “exception” when answer can’t be represented)

  0101.5
  + 0110.6
  1011.15?!!

A Basic ALU

```
+--------+--------+--------+--------+--------+--------+
<table>
<thead>
<tr>
<th>A</th>
<th>N</th>
<th>B</th>
<th>Zero</th>
<th>Result</th>
<th>Overflow</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>CarryOut</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

- **ALU Control Lines (ALUop)**
  - 000: And
  - 001: Or
  - 010: Add
  - 011: Subtract
  - 100: Set-on-less-than

General idea: Build for 1-bit numbers and then extend for n-bits?

Some basics of digital logic

1. **AND gate (I \& x b)**

```
\[ i \land (x \land b) \]
```

2. **OR gate (I + x b)**

```
\[ i \lor (x \lor b) \]
```

3. **Inverter (I \neg x b)**

```
\[ \neg i \]
```

4. **Multiplexor (I \times x b)**

```
\[ i \times (x \times b) \]
```
1-bit ALU

- ALU Control Lines (ALUop) Function
  - 000 And
  - 001 Or
  - 001 Add

But how do we make the adder?

1-bit Full Adder

- This is also called a (3, 2) adder
- Half Adder: No CarryIn nor CarryOut
- Truth Table:

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>CarryIn</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

1-bit Full Adder: CarryOut

\[
\text{CarryOut} = (A \& B \& \text{CarryIn}) \text{ Or } (A \& \neg B \& \text{CarryIn}) \text{ Or } (A \& B \& \neg \text{CarryIn})
\]

1-bit Full Adder: Sum

\[
\text{Sum} = (A \& \neg B \& \text{CarryIn}) \text{ Or } (A \& B \& \text{CarryIn}) \text{ Or } (A \& B \& \text{CarryIn}) \text{ Or } (A \& B \& \text{CarryIn})
\]

32-bit ALU

- (ALUop) Function
  - 000 And
  - 001 Or
  - 11 Add

What about other operations
- sub S1, S2, S3: S1 = S2 - S3: Subtraction
- srl S1, S2, S3: S1 = (S1 << S2) and (S1 >> S3): Shift Right
- sll S1, S2, S3: S1 = (S1 << S2) and (S1 >> S3): Shift Left
- sgt S1, S2, S3: S1 = (S1 >> S2) and (S1 >> S3): Shift Greater Than
- sc S1, S2, S3: S1 = (S1 >> S2) and (S1 >> S3): Shift Cutoff
- sct S1, S2, S3: S1 = (S1 >> S2) and (S1 >> S3): Shift Cutoff with Trap
- sctb S1, S2, S3: S1 = (S1 >> S2) and (S1 >> S3): Shift Cutoff with Trap and Branch
- scct S1, S2, S3: S1 = (S1 >> S2) and (S1 >> S3): Shift Cutoff with Trap and Trap
- scctb S1, S2, S3: S1 = (S1 >> S2) and (S1 >> S3): Shift Cutoff with Trap and Trap and Branch

32-bit ALU

- Keep in mind the following:
  - (A \& B) is the same as: A + (\neg B)
  - 2's Complement negation: Take the inverse of every bit and add 1
  - Bit-wise inverse of B is B:
    - A \& B + A + B = A + B + 1

Binvert provides the negation
### 32-bit ALU

- **Overflow Detection Logic:**
  - Carry into MSB \( = \) Carry out of MSB
    - For a 32-bit ALU Overflow = Carry\(N - 1\) \( \times \) Carry\(N - 1\)
  - \(A\), \(B\) = 1-bit ALU
    - Result0 = \(X \times Y\)
      - \(X = 0 \quad Y = 0\)
      - \(X = 0 \quad Y = 1\)
      - \(X = 1 \quad Y = 0\)
      - \(X = 1 \quad Y = 1\)
    - CarryOut0 = \(X \oplus Y\)
  - \(A\), \(B\) = 1-bit ALU
    - Result1 = \(X \times Y\)
    - CarryOut1 = \(X \oplus Y\)
  - \(A\), \(B\) = 1-bit ALU
    - Result2 = \(X \times Y\)
    - CarryOut2 = \(X \oplus Y\)
  - \(A\), \(B\) = 1-bit ALU
    - Result3 = \(X \times Y\)
    - CarryOut3 = \(X \oplus Y\)

- **Zero Detection Logic:**
  - Any non-zero input to the NOR gate will cause its output to be zero

### 32-bit ALU: Special conditions

- **Zero Detection Logic:**
  - Any non-zero input to the NOR gate will cause its output to be zero

- **Overflow Detection Logic:**
  - Carry into MSB = Carry out of MSB
    - For a 32-bit ALU Overflow = Carry\(N - 1\) \( \times \) Carry\(N - 1\)

### 32-bit ALU: Special conditions

- **32-bit ALU: sll**
  - All instructions
  - If \((a - b) < 0\) result = 1, else result = 0.
  - If \((a - b) < 0\) result = 1, else result = 0.

- **32-bit ALU: slt**
  - Subtract
  - Use sign bit
    - Move to bit 0 of result
    - All other bits zero

- **32-bit ALU: Special conditions**
  - Thus MSB block has special logic to generate
    - Set line (sign bit)
    - Overflow line

- **32-bit ALU**
  - Notice control lines:
    - 000 = and
    - 001 = or
    - 010 = add
    - 110 = subtract
    - 111 = sll

- **Zero is a 1 when the result is zero?**

- But what about performance?
**32 bit ALU**

- We can build an ALU to support the MIPS instruction set
  - key idea: use multiplexor to select the output we want
  - we can efficiently perform subtraction using two’s complement
  - we can replicate a 1-bit ALU to produce a 32-bit ALU
- Important points about hardware
  - all of the gates are always working
  - the speed of a gate is affected by the number of inputs to the gate
  - the speed of a circuit is affected by the number of gates in series
    (on the “critical path” or the “deepest level of logic”)
- Our primary focus: comprehension, however,
  - Clever changes to organization can improve performance
    (similar to using better algorithms in software)
  - we’ll look at two examples for addition and multiplication

**Performance**

**Ideal (IS) versus Reality (EE)**

- When input 0 → 1, output 1 → 0 but NOT instantly
  - Output goes 1 → 0; output voltage goes from Vdd (5V) to 0
- When input 1 → 0, output 0 → 1 but NOT instantly
  - Output goes 0 → 1; output voltage goes from 0 to Vdd (5V)
- Voltage does not like to change instantaneously

**Series Connection**

- Total Propagation Delay = Sum of individual delays = d1 + d2
- Capacitance C1 has two components:
  - Capacitance of the wire connecting the two gates
  - Input capacitance of the second inverter

**Performance: Calculating Delays**

- Sum delays along serial paths
- Delay (Vin → V2) = Delay (V2 → V3)
- Delay (Vin → V2) = Delay (V2 → V1) + Delay (V1 → V2)
- Delay (Vn → V3) = Delay (Vn → V1) + Delay (V1 → V3)
- Critical Path = The longest among the N parallel paths
  - C1 = Wire C + Cin of Gate 2 + Cin of Gate 3

**Performance: Storage elements**

- Storage element: D flip flop with negative edge triggered

**Performance: Synchronous logic**

- All storage elements are clocked by the same clock edge
- The combination logic block’s:
  - Inputs are updated at each clock tick
  - All outputs MUST be stable before the next clock tick
Performance: Critical Path

- Critical path: the slowest path between any two storage devices
- Cycle time is a function of the critical path
  - must be greater than:
    - Clock-to-Q + Longest Path through the Combination Logic + Setup

Cycle Time: Thumb rules

- Reduce the number of gate levels
- Pay attention to loading
  - One gate driving many gates is a bad idea
  - Avoid using a small gate to drive a long wire
- Use multiple stages to drive large load

Ripple Carry Adders

- Is there more than one way to do addition?
  - two extremes: ripple carry and sum-of-products

Can you see the ripple? How could you get rid of it?

\[
\begin{align*}
  c_i &= b_i \oplus a_i \oplus \overline{b_i} \\
  c_0 &= b_0 \oplus a_0 \oplus \overline{b_0} \\
  c_2 &= b_2 \oplus a_2 \oplus \overline{b_2} \\
  c_3 &= b_3 \oplus a_3 \oplus \overline{b_3} \\
  c_i &= b_i \oplus a_i \oplus \overline{b_i}
\end{align*}
\]

Not feasible! Why?

Clock Skew

- The worst case scenario for cycle time consideration:
  - The input register sees CLK1
  - The output register sees CLK2
  - Cycle Time = Clock-to-Q + Longest Delay + Setup + Clock Skew

Back to ALUs

- The adder we just built is called a "Ripple Carry Adder"
  - The carry bit may have to propagate from LSB to MSB
  - Worst case delay for an n-bit RC adder: 2n-gate delay

Carrier

- Example:
  - Single gate delay = 0.02 ns (inverter "speed" of 50 GHz)
  - 32-bit adder => 64 gate delay => 1.28 ms delay => maximum clock of 789 MHz.

Carry Look Ahead Adders

- An approach in-between our two extremes

Motivation:

- If we didn’t know the value of carry-in, what could we do?
  - When would we always generate a carry?
    - \( g = a \land b \)
  - When would we propagate the carry?
    - \( p = a \lor b \)

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b )</td>
<td>( a )</td>
<td>( g )</td>
</tr>
</tbody>
</table>
| 0 | 0 | 0 | 0 | \( g \lor (a \land b) \)
| 1 | 0 | 1 | 1 | \( 1 \lor (a \land b) \)
| 0 | 1 | 1 | 1 | \( 1 \lor (a \land b) \)
| 1 | 1 | 1 | 1 | \( 1 \lor (a \land b) \)

Propagate Carry: \( \text{CarryOut} = \text{CarryIn} \)

Generate Carry: \( \text{CarryOut} = 1 \) (independent of CarryIn)
Carry Look Ahead Adders

The Propagate and Generate machinery.

\[ \text{Prop} = P_i \oplus C_{i-1} \]
\[ \text{Gen} = P_i \cdot C_{i-1} \]

Worst case delay of 1-gate.

Carry Look Ahead Adders

The Generation of the CarryOut.

The delay (and size) still grows with number of bits.

Carry Look Ahead Adders

The Generation of the Result.

\[ \text{Sum}_i = P_i \oplus C_{i-1} \]
\[ \text{Prop}_i = A_i \cdot B_i \]

Carry Look Ahead Adders

- It is very expensive to build a "full" carry look ahead adder
  - Just imagine the length of the equation for \( C_{31} \)
- Common practice:
  - Connect several N-bit Lookahead Adders to form a big adder
  - Example: connect four 8-bit carry look ahead adders to form a 32-bit partial carry lookahead adder

Carry Look Ahead Adders

What did we cover today?

- Last pieces of ISA class
- Performance: how to quantify
- Binary representation: integers, positive and negative
- Basic ALU design
- 1-bit addition
- Handling the carry
- Carry look ahead
- Subtraction
- Set on less than
- Condition codes such as overflow, zero
- Performance: Cycle time, number of gates etc.

Next class
Multiplication, Division, Floating point numbers
Rest of Chapter 4 from the text
Remember quizzes are surprises, and based on hw!