Integer Multiplication (1/2)

° Paper and pencil example (unsigned):

\[
\begin{align*}
\text{Multiplicand} & \quad 1000 \quad 8 \\
\text{Multiplier} & \quad x1001 \quad 9 \\
\hline
1000 & \\
0000 & \\
0000 & \\
+1000 & \\
\hline
01001000 & 
\end{align*}
\]

• m digits \times n digits = m + n digit product
Integer Multiplication (2/2)

° Register product:
  • 32-bit value x 32-bit value = 64-bit value

° Syntax of Multiplication (signed):
  • `mult register1, register2`
  • `puts product upper half in hi, lower half in lo`
  • `mfhi` & `mflo` to move from hi, lo to another register
Integer Division (1/2)

° Paper and pencil example (unsigned):

\[
\begin{array}{c|c c c c c}
\text{Divisor} & 1000 & | & 1001010 & \text{Dividend} \\
\hline
\text{Quotient} & 1001 & \quad \text{Remainder} & 10 \\
\hline
-1000 & 10 & 101 & 1010 & -1000 & 10 \\
\hline
\end{array}
\]

\( \text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder} \)
Integer Division (2/2)

° Division (signed):
  • `div` register1, register2
  • remainder in hi, quotient in lo

° Implements C division (/) and modulo (%)

° Example in C:
  ```
  a = c / d;
  b = c % d;
  ```

° in MIPS:
  ```
  a←$s0; b←$s1; c←$s2; d←$s3
  
  div  $s2,$s3  # lo=c/d, hi=c%d
  mflo $s0      # get quotient
  mfhi $s1     # get remainder
  ```
Unsigned Instructions & Overflow

° MIPS also has versions of `mult`, `div` for unsigned operands:

`multu`
`divu`

° MIPS does not check overflow on ANY signed/unsigned multiply or divide instruction

• Up to the software to check `hi`
Floating Point Representation (1/2)

° Normal format: $+1.xxxxxxxxxxx_2 \times 2^{yyyy}_2$

° Multiple of Word Size (32 bits)

31 30  23 22  0

S | Exponent | Significand

1 bit 8 bits 23 bits

° Represent numbers as small as $2.0 \times 10^{-38}$ to as large as $2.0 \times 10^{38}$
Floating Point Representation (2/2)

° What if result too large? (> $2.0 \times 10^{38}$ )
  • **Overflow!**
    • Overflow $\Rightarrow$ Exponent larger than represented in 8-bit Exponent field

° What if result too small? (>0, < $2.0 \times 10^{-38}$ )
  • **Underflow!**
    • Underflow $\Rightarrow$ Negative exponent larger than represented in 8-bit Exponent field

° How to reduce chances of overflow or underflow? => Double Precision (64 bits)
IEEE 754 Floating Point Standard (1/2)

° Sign bit: 1 means negative
  0 means positive

° Significand:
  • leading 1 implicit => normalized numbers

° Note: 0 has no leading 1, so reserve
  exponent value 0 just for number 0

° Biased Notation
  • Subtract 127 from Exponent field to get
    actual value for exponent
IEEE 754 Floating Point Standard (2/2)

° Summary (single precision):

\[
\begin{array}{ccc}
31 & 30 & 23 \quad 22 \\
\text{S} & \text{Exponent} & \text{Significand} \\
1 \text{ bit} & 8 \text{ bits} & 23 \text{ bits}
\end{array}
\]

° \((-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}\)
Example: Converting Binary FP to Decimal

\[
\begin{array}{cccccc}
0 & 0110 & 1000 & 101 & 0101 & 0100 & 0011 & 0100 & 0010 \\
\end{array}
\]

° Sign: 0 => positive

° Exponent:
  • \(0110 \ 1000_{\text{two}} = 104_{\text{ten}}\)
  • Bias adjustment: 104 - 127 = -23

° Significand:

\[
1 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} + \ldots
= 1 + 2^{-1} + 2^{-3} + 2^{-5} + 2^{-7} + 2^{-9} + 2^{-14} + 2^{-15} + 2^{-17} + 2^{-22}
= 1.0_{\text{ten}} + 0.666115_{\text{ten}}
\]

° Represents: \(1.666115_{\text{ten}} \times 2^{-23} \sim 1.986 \times 10^{-7}\)

(about 2/10,000,000)
Converting Decimal to FP

° Show MIPS representation of -0.75

• $-0.75 = -\frac{3}{4}$
• $-11_{\text{two}}/100_{\text{two}} = -0.11_{\text{two}}$
• Normalized to $-1.1_{\text{two}} \times 2^{-1}$
• $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$
• $(-1)^1 \times (1 + .100\ 0000\ ...\ 0000) \times 2^{(126-127)}$

```
1 0111 1110 100 0000 0000 0000 0000 0000
```
**Special Numbers**

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Significand</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>nonzero</td>
<td>Denoms</td>
</tr>
<tr>
<td>1-254</td>
<td>anything</td>
<td>+/- fl. pt. #</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>+/- ∞</td>
</tr>
<tr>
<td>255</td>
<td>nonzero</td>
<td>NaN</td>
</tr>
</tbody>
</table>
FP Addition & Subtraction

° can’t just add significands

° Algorithm
  • De-normalize to match larger exponent
  • Add significands to get resulting one
  • Normalize (& check for under/overflow)
  • Round if needed (may need to renormalize)

° If signs ≠, do a subtract. (Subtract similar)
  • If signs ≠ for add (or = for sub), what’s ans sign?
MIPS Floating Point Architecture

° Separate floating point instructions:
  • Single Precision:
    add.s, sub.s, mul.s, div.s
  • Double Precision:
    add.d, sub.d, mul.d, div.d

° far more complicated than integer
  • Can take much longer to execute
The NOT logic gate is an inverter.

Logical function: \( A' \) or \( \overline{A} \)

<table>
<thead>
<tr>
<th>Input A</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
NAND Logic Gate

- Logical function “NAND”
- \( \overline{AB} \) alternatively \((AB)'\)

<table>
<thead>
<tr>
<th>Input A</th>
<th>Input B</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
AND Logic Gate

° Logical function “AND”

° AB
OR Logic Gate

- Logical function “OR”
- \[ A + B \]
NOR Logic Gate

- Logical function “NOT OR”
- \((A + B)\) alternatively \((A + B)'\)

<table>
<thead>
<tr>
<th>Input A</th>
<th>Input B</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<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>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
**XOR Logic Gate**

- **Logical function “Exclusive OR”**
- \( \overline{A} B + A \overline{B} \) alternatively \( A'B + AB' \)

<table>
<thead>
<tr>
<th>Input A</th>
<th>Input B</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
# Truth Tables

Define the output for any given combination of inputs

<table>
<thead>
<tr>
<th>Input A</th>
<th>Input B</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input A</th>
<th>Input B</th>
<th>Input C</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Creating a Digital Logic Design from a Truth Table

° Canonical Sum of Products form

• Create an input line for each input variable, and branch off another line with a NOT gate to form the complement of the variable.

• You will need as many AND gates as there are “1”s in the truth table. The inputs to each AND gate are the input variables or their complements - as indicated by the truth table entries.

• For example if x=0, y=1, and z=0 and OUTPUT=1, you will need x’, y, and z’ as inputs into one of the AND gates.

• You will need one OR gate. All of the outputs of the AND gates will go into the OR gate. The output of the OR gate is the output function for the circuit.
Function 2 Circuit

\[ f (A,B,C) = A' + B' + A' B C \]
Simplified solutions aren’t always unique

\[ f(x,y,z) = x'y'z' + x'y'z + x'y'z + x'yz' + x'yz + xyz \]

\[ = x'z' + x'y' + yz \]

\[ = x'z' + y'z' + yz + xz \]
Examples of Combinational Digital Logic Functions

- Multiplexers
- Decoders
- Parity Generators
- Adders
  - Half adders
  - Full adders
  - 32-bit adders
- Shifters
- Comparators
- Arithmetic Logic Units
Half Adder

° Consists of:
  • 2 inputs
  • 2 outputs
    - Sum
    - Carry

° Used for basic integer addition (1 bit)

<table>
<thead>
<tr>
<th>Input A</th>
<th>Input B</th>
<th>Sum</th>
<th>Carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
# Full Adder

<table>
<thead>
<tr>
<th>Input A</th>
<th>Input B</th>
<th>Carry In</th>
<th>Sum</th>
<th>Carry Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Full Adder

A
B

Carry In

Carry Out

Sum
N-bit adders
Multiplexers

4 input multiplexer

D0
D1
D2
D3
A
B
Out
Decoders

2 to 4 decoder

A

B

D_0

D_1

D_2

D_3
A Very Simple ALU

° Performs four functions:

A .AND. B   (A & B)  //bitwise AND
A .OR. B     (A | B)  //bitwise OR
NOT.B        (~ B)    //bitwise NOT
A + B         (A + B)  //addition operator

° Operates on single bits “A” and “B”
Block Diagram for ALU

Logic Unit (AND, OR, NOT)

A
B

Enable A
Enable B

2 to 4 Decoder

F1
F0

En OR
En AND
En NOT
En ADD

Full Adder

A
B

Sum

Carry In

Output

Carry Out
Latch Operation

<table>
<thead>
<tr>
<th>Set</th>
<th>Reset</th>
<th>Q</th>
<th>( \overline{Q} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Q</td>
<td>( \overline{Q} )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>undefined</td>
<td>undefined</td>
</tr>
</tbody>
</table>
Clocked SR Latches

Set

Clock

Reset

Memory Element

Q

\( Q \)
The D Latch
- A Variation on the SR Latch -

D
Q

Clock

Memory Element

Q
D Latches and D Flip-Flops

D Latch

DQ

CK

level triggered

D Flip-Flop

DQ

> CK

edge triggered
Register Bit "3"

Register Bit "2"

Register Bit "1"

Register Bit "0"

Clock
Register Files

CLOCK

A_0

A_1

D_0

D_1

D_2

D_3

Clock

Input Bit "3"

Input Bit "2"

Input Bit "1"

Input Bit "0"
Memory - Bit Slice

- Cells connected to form 1 bit position
- Word Select gates one latch from address lines
- Note it selects Reads also
- B (and B not) set by R/W, Data In and BitSelect
Bit Slice can Become Module

- Basically bit slice is a X1 memory

- Next
16 X 1 as 4 X 4 Array

- Two decoders
  - Row
  - Column

- Address just broken up

- Not visible from outside
Single cycle and control unit
Single cycle plus jump
Example of execution

Instr: add $s0, $a1, $t7
Field: op rs rt rd shamt funct
M. Code: 000000 00101 01111 10000 00000 100000
PC: 0 x 4000
Extra hardware for multicycle datapath
Multicycle datapath
Complete multicycle datapath
# Multicycle Datapath: the 5 steps

<table>
<thead>
<tr>
<th>Step name</th>
<th>Action for R-type instructions</th>
<th>Action for memory-reference instructions</th>
<th>Action for branches</th>
<th>Action for jumps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction fetch</td>
<td></td>
<td>IR &lt;= Memory[PC]</td>
<td></td>
<td>PC &lt;= PC + 4</td>
</tr>
<tr>
<td>Instruction decode/register fetch</td>
<td></td>
<td>A &lt;= Reg [IR[25:21]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>B &lt;= Reg [IR[20:16]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALUOut &lt;= PC + (sign-extend (IR[15:0]) &lt;&lt; 2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution, address computation, branch/jump completion</td>
<td>ALUOut &lt;= A op B</td>
<td>ALUOut &lt;= A + sign-extend (IR[15:0])</td>
<td>If (A == B)</td>
<td>PC &lt;= ALUOut</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PC &lt;= {PC [31:28], (IR[25:0]), 2'b00}</td>
</tr>
<tr>
<td>Memory access or R-type completion</td>
<td>Reg [IR[15:11]] &lt;= ALUOut</td>
<td>Load: MDR &lt;= Memory[ALUOut] or Store: Memory [ALUOut] &lt;= B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory read completion</td>
<td></td>
<td>Load: Reg[IR[20:16]] &lt;= MDR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Every instruction must take same number of steps, also called pipeline “stages”, so some will go idle sometimes.
Pipeline Stages

1. Instruction Fetch
2. Decode/Register Read
3. Execute
4. Memory
5. Write Back

IFtch | Dcd | Exec | Mem | WB

I$ | Reg | ALU | D$ | Reg

PC \rightarrow \text{instruction memory} \rightarrow \text{registers} \rightarrow \text{ALU} \rightarrow \text{Data memory}
Graphical Pipeline Representation

Time (clock cycles)

<table>
<thead>
<tr>
<th>Instruction Order</th>
<th>I$</th>
<th>ALU</th>
<th>Reg</th>
<th>D$</th>
<th>Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>I$</td>
<td>IS</td>
<td>Reg</td>
<td></td>
<td>D$</td>
</tr>
<tr>
<td>Add</td>
<td>IS</td>
<td>IS</td>
<td>ALU</td>
<td>Reg</td>
<td>D$</td>
</tr>
<tr>
<td>Store</td>
<td>IS</td>
<td>IS</td>
<td>ALU</td>
<td>Reg</td>
<td>D$</td>
</tr>
<tr>
<td>Sub</td>
<td>IS</td>
<td>IS</td>
<td>ALU</td>
<td>Reg</td>
<td>D$</td>
</tr>
<tr>
<td>Or</td>
<td>IS</td>
<td>IS</td>
<td>ALU</td>
<td>Reg</td>
<td>D$</td>
</tr>
</tbody>
</table>
Limits to pipelining

° **Hazards** prevent next instruction from executing during its designated clock cycle

- **Structural hazards**: HW cannot support this combination of instructions (single person to fold and put clothes away)

- **Control hazards**: Pipelining of branches & other instructions **stall** the pipeline until the hazard; “**bubbles**” in the pipeline

- **Data hazards**: Instruction depends on result of prior instruction still in the pipeline (missing sock)
Structural Hazard #1: Single Memory (1/2)

Read same memory twice in same clock cycle
Structural Hazard #1: Single Memory (2/2)

° Solution:

• infeasible and inefficient to create second memory

• so handle this by having two Level 1 Caches (a temporary smaller [of usually most recently used] copy of memory)

• have both an L1 Instruction Cache and an L1 Data Cache

• need more complex hardware to control when both caches miss
Structural Hazard #2: Registers (1/2)

Can’t read and write to registers simultaneously
Structural Hazard #2: Registers (2/2)

° Fact: Register access is *VERY* fast: takes less than half the time of ALU stage

° Solution: introduce convention
  • always Write to Registers during first half of each clock cycle
  • always Read from Registers during second half of each clock cycle
  • Result: can perform Read and Write during same clock cycle
Control Hazard: Branching

Where do we do the compare for the branch?
Control Hazard: Branching

Optimization #1:

• move asynchronous comparator up to Stage 2

• Identify branches asap

• Benefit: since branch is complete in Stage 2, only one unnecessary instruction is fetched, so only one no-op is needed

• Side Note: This means that branches are idle in Stages 3, 4 and 5.
Control Hazard: Branching

• Optimization #2: Redefine branches
  • Old definition: if we take the branch, none of the instructions after the branch get executed by accident
  • New definition: whether or not we take the branch, the single instruction immediately following the branch gets executed (called the branch-delay slot)

• The term “Delayed Branch” means we always execute inst after branch
Example: Nondelayed vs. Delayed Branch

**Nondelayed Branch**
- `or   $8, $9 ,$10`
- `add $1 ,$2,$3`
- `sub $4, $5,$6`
- `beq $1, $4, Exit`
- `xor $10, $1,$11`

**Delayed Branch**
- `add $1 ,$2,$3`
- `sub $4, $5,$6`
- `beq $1, $4, Exit`
- `or   $8, $9 ,$10`
- `xor $10, $1,$11`

**Exit:**
- `Exit:`
Data Hazards

° Consider the following sequence of instructions

\[
\begin{align*}
&\text{add } \texttt{t0}, \texttt{t1}, \texttt{t2} \\
&\text{sub } \texttt{t4}, \texttt{t0}, \texttt{t3} \\
&\text{and } \texttt{t5}, \texttt{t0}, \texttt{t6} \\
&\text{or } \texttt{t7}, \texttt{t0}, \texttt{t8} \\
&\text{xor } \texttt{t9}, \texttt{t0}, \texttt{t10}
\end{align*}
\]
Dependencies backwards in time are hazards

**Data Hazards (2/2)**

- add $t0, $t1, $t2
- sub $t4, $t0, $t3
- and $t5, $t0, $t6
- or $t7, $t0, $t8
- xor $t9, $t0, $t10

Time (clock cycles)
Data Hazard Solution: Forwarding

• **Forward** result from one stage to another

```
add $t0, $t1, $t2
sub $t4, $t0, $t3
and $t5, $t0, $t6
or  $t7, $t0, $t8
xor $t9, $t0, $t10
```

“or” hazard solved by register hardware
Data Hazard: Loads

• Dependencies backwards in time are hazards

\[
\text{lw } \$t0,0(\$t1) \\
\text{sub } \$t3,\$t0,\$t2
\]

• Can’t solve with forwarding
• Must stall instruction dependent on load, then forward (more hardware)
Data Hazard: Loads

- **Hardware** must stall pipeline
- Called “**interlock**”

```assembly
lw $t0, 0($t1)
sub $t3,$t0,$t2
and $t5,$t0,$t4
or $t7,$t0,$t6
```
Data Hazard: Loads

° Instruction slot after a load is called “load delay slot”

° If that instruction uses the result of the load, then the hardware interlock will stall it for one cycle.

° If the compiler puts an unrelated instruction in that slot, then no stall

° Letting the hardware stall the instruction in the delay slot is equivalent to putting a nop in the slot (except the latter uses more code space)
Data Hazard: Loads

° Stall is equivalent to nop

lw $t0, 0($t1)

nop

sub $t3,$t0,$t2

and $t5,$t0,$t4

or $t7,$t0,$t6