Other Issues

- support for procedures (Refer to section 3.6), stacks, frames, recursion
- manipulating strings and pointers
- Interrupts, exceptions, system calls and conventions
- Register use convention

<table>
<thead>
<tr>
<th>Name</th>
<th>Register number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>the constant value 0</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>values for results and expression evaluation</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>arguments</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>temporaries</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>saved</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>more temporaries</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
</tr>
</tbody>
</table>

Stack Manipulation

- Register $29 is used as stack pointer
- Stack grows from high address to low address
- Stack pointer should point to the last filled address
- Once entries are removed, stack pointer should be adjusted

Frame Pointer

- Stores the last address for the last frame
- When completing a subroutine, frame address can be used as the starting stack pointer value

How about larger constants?

- We’d like to be able to load a 32 bit constant into a register
- Must use two instructions, new "load upper immediate" instruction
  ```
  lui $t0, 1010101010101010
  ```
  filled with zeros
- Then must get the lower order bits right, i.e.,
  ```
  ori $t0, $t0, 1010101010101010
  ```

Alternative Architectures

- Design alternative:
  - provide more powerful operations
  - goal is to reduce number of instructions executed
  - danger is a slower cycle time and/or a higher CPI
- Sometimes referred to as “RISC vs. CISC”
  - virtually all new instruction sets since 1982 have been RISC
  - VAX: minimize code size, make assembly language easy
    - instructions from 1 to 54 bytes long!
- We’ll look at PowerPC and 80x86

PowerPC

- Indexed addressing
  - example: 1w $t1,.$a0+$s3 #t1=Memory[$a0+$s3]
  - What do we have to do in MIPS?
- Update addressing
  - update a register as part of load (for marching through arrays)
    - example: 1wu $t0,.4(.$a3) #t0=Memory[.a3+4] ;a3=$a3+4
    - What do we have to do in MIPS?
  - Others:
    - load multiple/store multiple
    - a special counter register “bc Loop”
      ```
      decrement counter, if not 0 goto loop
      ```
**80x86**

- 1978: The Intel 8086 is announced (16-bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, +instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)
- 1997: MMX is added

"This history illustrates the impact of the "golden handcuffs" of compatibility
adding new features as someone might add clothing to a packed bag"

"an architecture that is difficult to explain and impossible to love"

---

**A dominant architecture: 80x86**

- Complexity:
  - Instructions from 1 to 17 bytes long
  - One operand must act as both a source and destination
  - One operand can come from memory
  - Complex addressing modes
    - e.g., "base or scaled index with 8 or 32 bit displacement"
- Saving grace:
  - The most frequently used instructions are not too difficult to build
  - Compilers avoid the portions of the architecture that are slow

"what the 80x86 lacks in style is made up in quantity, making it beautiful from the right perspective."

---

**Registers in 80xY86 Architecture**

- EAX, EBX, ECX, EDX: General purpose registers
- ES, DS, CS: Segments
- SI, DI: Indirect index registers
- BP: Base pointer
- CS: Code segment
- DS: Data segment
- FS, GS: Additional data segments

---

**Examples of non-numeric instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JNZ</td>
<td>Jump if not zero</td>
</tr>
<tr>
<td>JMP</td>
<td>Jump to memory</td>
</tr>
<tr>
<td>POP EAX</td>
<td>Pop EAX</td>
</tr>
<tr>
<td>ADD EAX, IMM32</td>
<td>Add to EAX</td>
</tr>
<tr>
<td>TEST EDX, 64</td>
<td>Test condition code (flags) with EDX &amp; 65</td>
</tr>
<tr>
<td>MOVEL</td>
<td>Move long</td>
</tr>
</tbody>
</table>

---

**Instruction Encoding**

- Instruction complexity is only one variable
  - Lower instruction count vs. higher CPI / lower clock rate
- Design Principles:
  - Simplicity favors regularity
  - Smaller is faster
  - Good design demands compromise
  - Make the common case fast
- Instruction set architecture
  - A very important abstraction indeed!
Arithmetic

- Where we've been:
  - Performance (seconds, cycles, instructions)
  - Abstractions:
    - Instruction Set Architecture
    - Assembly Language and Machine Language
- What's up ahead:
  - Implementing the Architecture

Numbers

- Bits are just bits (no inherent meaning)
  - conventions define relationship between bits and numbers
- Binary numbers (base 2)
  - 0000 0001 0010 0101 0110 0111 1000 1001...
  - decimal: 0...2^n-1
- Of course it gets more complicated:
  - numbers are finite (overflow)
  - fractions and real numbers
  - e.g., no MIPS sub instruction; addi can add a negative number
- How do we represent negative numbers?
  - i.e., which bit patterns will represent which numbers?

Possible Representations

- Sign Magnitude: One's Complement Two's Complement
  - 000 = +0 000 = +0 000 = +0
  - 001 = +1 001 = +1 001 = +1
  - 010 = +2 010 = +2 010 = +2
  - 011 = +3 011 = +3 011 = +3
  - 100 = -0 100 = -3 100 = -4
  - 101 = -1 101 = -2 101 = -3
  - 110 = -2 110 = -1 110 = -2
  - 111 = -3 111 = -0 111 = -1
- Issues: balance, number of zeros, ease of operations
- Which one is best? Why?

MIPS

- 32 bit signed numbers:
  - 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
  - ...0111 1111 1111 1111 1111 1111 1111 1101 two = +2,147,483,646 ten
  - 0111 1111 1111 1111 1111 1111 1111 1110 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
  - ...1111 1111 1111 1111 1111 1111 1111 1111 two = -1 ten
- Issues: balance, number of zeros, ease of operations
- Which one is best? Why?

Two's Complement Operations

- Negating a two's complement number: invert all bits and add 1
  - remember: “negate” and “invert” are quite different!
- Converting n bit numbers into numbers with more than n bits:
  - MIPS 16 bit immediate gets converted to 32 bits for arithmetic
  - copy the most significant bit (the sign bit) into the other bits
  - 0010 -> 0000 0010
  - 1010 -> 1111 1010
- “sign extension” (lbu vs. lb)

Addition & Subtraction

- Just like in grade school (carry/borrow 1s)
  - 0111 + 0110 = 1101
  - 0111 + 0110 = 1101
  - 0111 + 0110 = 1101
- Two's complement operations easy
  - subtraction using addition of negative numbers
  - 0111 + 1010
- 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
- Overflow term is somewhat misleading.
  - 1000
  - does not mean a carry “overflowed”
One B Adder

- Takes three input bits and generates two output bits
- Multiple bits can be cascaded

Adder Boolean Algebra

- $A \land B \land C \land S$
- $0 \ 0 \ 0 \ 0 \ 0$
- $0 \ 0 \ 1 \ 0 \ 1$
- $0 \ 1 \ 0 \ 0 \ 1$
- $0 \ 1 \ 0 \ 0 \ 0$
- $1 \ 0 \ 0 \ 0 \ 0$
- $1 \ 0 \ 1 \ 1 \ 0$
- $1 \ 1 \ 0 \ 1 \ 0$
- $1 \ 1 \ 1 \ 1 \ 1$

Detecting Overflow

- No overflow when adding a positive and a negative number
- No overflow when signs are the same for subtraction
- Overflow occurs when the value affects the sign:
  - or, adding two negatives gives a positive
  - or, subtracting a negative from a positive and get a negative
  - or, subtracting a positive from a negative and get a positive
- Consider the operations $A + B$, and $A - B$
  - Can overflow occur if $B$ is 0?
  - Can overflow occur if $A$ is 0?

Effects of Overflow

- An exception (interrupt) occurs
  - Control jumps to predefined address for exception
  - Interrupted address is saved for possible resumption
- Details based on software system / language
  - example: flight control vs. homework assignment
- Don't always want to detect overflow
  - new MIPS instructions: addu, addiu, subu

Real Design

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<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>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

An ALU (arithmetic logic unit)

- Let's build an ALU to support the andi and ori instructions
  - we'll just build a 1 bit ALU, and use 32 of them

Possible Implementation (sum-of-products):
Different Implementations

- Not easy to decide the “best” way to build something
  - Don’t want too many inputs to a single gate
  - Don’t want to have to go through too many gates
  - For our purposes, ease of comprehension is important
- Let’s look at a 1-bit ALU for addition:

\[ c_{out} = a + b + c_{in} \]
\[ sum = a \oplus b \oplus c_{in} \]

- How could we build a 1-bit ALU for addition, and, and or?
- How could we build a 32-bit ALU?

Building a 32 bit ALU

What about subtraction (a – b)?

- Two’s complement approach: just negate b and add.
- How do we negate?
- A very clever solution:

Tailoring the ALU to the MIPS

- Need to support the set-on-less-than instruction (slt)
  - Remember: slt is an arithmetic instruction
  - Produces a 1 if rs < rt and 0 otherwise
  - Use subtraction: (a-b) < 0 implies a < b
- Need to support test for equality (beq $t5, $t6, $t7)
  - Use subtraction: (a-b) = 0 implies a = b

Supporting slt

- Can we figure out the idea?

Test for equality

- Notice control lines:
  - 000 = and
  - 001 = or
  - 010 = add
  - 110 = subtract
  - 111 = slt

(Note: zero is a 1 when the result is zero)