ECE472, FA11: Lecture 3
Power Dissipation

Has been > doubling every 2 years

Has to stay ~constant
Power is a Major Problem

Power delivery and dissipation will be prohibitive

EECS141  Courtesy, Intel
Power Density

Power density too high to keep junctions at low temp

EECS141
Power Derivation

• $P = I \cdot V \Rightarrow I = C \cdot \frac{dV}{dt}$
• $P = C \cdot V \cdot \frac{dV}{dt}$
• $P \cdot dt = C \cdot V \cdot dV$ (INTEGRATE)
• $P \cdot t = \frac{1}{2} \cdot C \cdot V^2$
• $P = \frac{1}{2} \cdot f \cdot C \cdot V^2$
• $E = \frac{P}{t} = \frac{1}{2} \cdot C \cdot V^2$

• $f = \text{seconds/cycle}$
• Execution Time = $IC \cdot CPI \cdot \frac{1}{f}$

• NOTE: $f = \frac{1}{t_D} = \frac{1}{0.69} \cdot R \cdot C$; $I = \frac{1}{2} \cdot K_N \cdot \frac{W}{L} \cdot (V_{DD} - V_T)^2$
• $R \sim \frac{1}{V}$ ($R=V/I$)
• THEREFORE, $P \sim V^3$
Not Only Microprocessors

Cell Phone

Digital Cellular Market (Phones Shipped)

<table>
<thead>
<tr>
<th>Year</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>1996</td>
<td>48M</td>
</tr>
<tr>
<td>1997</td>
<td>86M</td>
</tr>
<tr>
<td>1998</td>
<td>162M</td>
</tr>
<tr>
<td>1999</td>
<td>260M</td>
</tr>
<tr>
<td>2000</td>
<td>435M</td>
</tr>
</tbody>
</table>

(data from Texas Instruments)
Productivity Trends

Complexity outpaces design productivity

Source: Sematech

EECS141  Courtesy, ITRS Roadmap
Cost of Integrated Circuits

- NRE (non-recurrent engineering) costs
  - design time and effort, mask generation
  - one-time cost factor

- Recurrent costs
  - silicon processing, packaging, test
  - proportional to volume
  - proportional to chip area
Mask Cost is Increasing

Cost [in $1000]

Year


45nm
65nm
90nm
0.13 μm
0.18 μm
0.25 μm

EECS141
Total Cost

- Cost per IC
  \[ \text{cost per IC} = \text{variable cost per IC} + \frac{\text{fixed cost}}{\text{volume}} \]

- Variable cost
  \[ \text{variable cost} = \frac{\text{cost of die} + \text{cost of die test} + \text{cost of packaging}}{\text{final test yield}} \]
Die Cost

Wafer

Single die

cost of die = \frac{\text{cost of wafer}}{\text{dies per wafer} \times \text{die yield}}

Going up to 12” (30cm)

From: http://www.amd.com
Yield

\[ Y = \frac{\text{No. of good chips per wafer}}{\text{Total number of chips per wafer}} \times 100\% \]

\[ \text{Die cost} = \frac{\text{Wafer cost}}{\text{Dies per wafer} \times \text{Die yield}} \]

\[ \text{Dies per wafer} = \frac{\pi \times (\text{wafer diameter}/2)^2}{\text{die area}} - \frac{\pi \times \text{wafer diameter}}{\sqrt{2} \times \text{die area}} \]
Defects

Yield = 0.25

Yield = 0.76

die yield = \left(1 + \frac{\text{defects per unit area} \times \text{die area}}{\alpha}\right)^{-\alpha}

\alpha is approximately 3

die cost = f(\text{die area})^4
## Some Examples (1994)

<table>
<thead>
<tr>
<th>Chip</th>
<th>Metal layers</th>
<th>Line width</th>
<th>Wafer cost</th>
<th>Def./cm²</th>
<th>Area mm²</th>
<th>Dies/wafer</th>
<th>Yield</th>
<th>Die cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>386DX</td>
<td>2</td>
<td>0.90</td>
<td>$900</td>
<td>1.0</td>
<td>43</td>
<td>360</td>
<td>71%</td>
<td>$4</td>
</tr>
<tr>
<td>486 DX2</td>
<td>3</td>
<td>0.80</td>
<td>$1200</td>
<td>1.0</td>
<td>81</td>
<td>181</td>
<td>54%</td>
<td>$12</td>
</tr>
<tr>
<td>Power PC 601</td>
<td>4</td>
<td>0.80</td>
<td>$1700</td>
<td>1.3</td>
<td>121</td>
<td>115</td>
<td>28%</td>
<td>$53</td>
</tr>
<tr>
<td>HP PA 7100</td>
<td>3</td>
<td>0.80</td>
<td>$1300</td>
<td>1.0</td>
<td>196</td>
<td>66</td>
<td>27%</td>
<td>$73</td>
</tr>
<tr>
<td>DEC Alpha</td>
<td>3</td>
<td>0.70</td>
<td>$1500</td>
<td>1.2</td>
<td>234</td>
<td>53</td>
<td>19%</td>
<td>$149</td>
</tr>
<tr>
<td>Super Sparc</td>
<td>3</td>
<td>0.70</td>
<td>$1700</td>
<td>1.6</td>
<td>256</td>
<td>48</td>
<td>13%</td>
<td>$272</td>
</tr>
<tr>
<td>Pentium</td>
<td>3</td>
<td>0.80</td>
<td>$1500</td>
<td>1.5</td>
<td>296</td>
<td>40</td>
<td>9%</td>
<td>$417</td>
</tr>
</tbody>
</table>
Cost per Transistor

Fabrication cost per transistor
Take-Aways

• Ex. Time = IC * CPI * Cycle Time
  – Cycle-Time = 1/f (clock frequency)

• P = \( \frac{1}{2} \times C \times V_{dd}^2 \times f \)

• E = P*t = \( \frac{1}{2} \times C \times V_{dd}^2 \) (NOTE: NOT a function of time)

• Describe how CMOS works

• \( t = \frac{1}{f} \Rightarrow 0.69 \times R \times C \)
  – R is transistor resistance
  – C is inverter capacitance

• \( R = \frac{V_{dd}}{I} \Rightarrow I \sim V_{dd}^2 \) (I is the transistor current)
  – \( R \sim \frac{1}{V_{dd}} \)

• \( P \sim V_{dd}^3 \)
Examples

- Latency metric: program execution time in seconds

  \[
  \text{CPUtime} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Cycles}}{\text{Program}} \cdot \frac{\text{Seconds}}{\text{Cycle}} \\
  = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Seconds}}{\text{Cycle}} \\
  = IC \cdot CPI \cdot CCT
  \]

- CCT decrease ‘X’-> power decreases by: ‘X or X^3’
  - Ex. Time increases by ‘X’

- How does N more cores improve things?
The Processor Market

![Graph showing the processor market from 1997 to 2007, categorizing by Cell Phones, PCs, and TVs. The graph illustrates a significant increase in processor demand, particularly for PCs and TVs, in recent years.]
Levels of Program Code

• High-level language
  – Level of abstraction closer to problem domain
  – Provides for productivity and portability

• Assembly language
  – Textual representation of instructions

• Hardware representation
  – Binary digits (bits)
  – Encoded instructions and data
Inside the Processor (CPU)

- Datapath: performs operations on data
- Control: sequences datapath, memory, ...
- Cache memory
  - Small fast SRAM memory for immediate access to data
Inside the Processor

• AMD Barcelona: 4 processor cores
Defining Performance

- Which airplane has the best performance?

![Graphs comparing airplane performance metrics](image-url)
CPU Clocking

• Operation of digital hardware governed by a constant-rate clock

- Clock period: duration of a clock cycle
  - e.g., 250ps = 0.25ns = 250×10^{-12}s

- Clock frequency (rate): cycles per second
  - e.g., 4.0GHz = 4000MHz = 4.0×10^9Hz
Instruction Count and CPI

Clock Cycles = Instruction Count × Cycles per Instruction

CPU Time = Instruction Count × CPI × Clock Cycle Time

= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}

• Instruction Count for a program
  – Determined by program, ISA and compiler

• Average cycles per instruction
  – Determined by CPU hardware
  – If different instructions have different CPI
    • Average CPI affected by instruction mix
**CPI Example**

- **Computer A:** Cycle Time = 250ps, CPI = 2.0
- **Computer B:** Cycle Time = 500ps, CPI = 1.2
- Same ISA
- Which is faster, and by how much?

<table>
<thead>
<tr>
<th>Computer</th>
<th>CPU Time</th>
<th>Instruction Count</th>
<th>CPI</th>
<th>Cycle Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$I \times 2.0 \times 250\text{ps}$</td>
<td>$I$</td>
<td>2.0</td>
<td>$250\text{ps}$</td>
</tr>
<tr>
<td>B</td>
<td>$I \times 1.2 \times 500\text{ps}$</td>
<td>$I$</td>
<td>1.2</td>
<td>$500\text{ps}$</td>
</tr>
</tbody>
</table>

\[
\frac{\text{CPU Time}_A}{\text{CPU Time}_B} = \frac{I \times 2.0 \times 250\text{ps}}{I \times 1.2 \times 500\text{ps}} = 1.2
\]

A is faster... by this much
CPI Example

- Alternative compiled code sequences using instructions in classes A, B, C

<table>
<thead>
<tr>
<th>Class</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPI for class</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>IC in sequence 1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>IC in sequence 2</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- **Sequence 1: IC = 5**
  - Clock Cycles
    - $= 2 \times 1 + 1 \times 2 + 2 \times 3$
    - $= 10$
  - Avg. CPI $= 10/5 = 2.0$

- **Sequence 2: IC = 6**
  - Clock Cycles
    - $= 4 \times 1 + 1 \times 2 + 1 \times 3$
    - $= 9$
  - Avg. CPI $= 9/6 = 1.5$
Performance Summary

The BIG Picture

\[
\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}
\]

- Performance depends on
  - Algorithm: affects IC, possibly CPI
  - Programming language: affects IC, CPI
  - Compiler: affects IC, CPI
  - Instruction set architecture: affects IC, CPI, \( T_c \)
Uniprocessor Performance

Constrained by power, instruction-level parallelism, memory latency
NVIDIA Fermi
3 x $10^9$ Transistors, 512 “cores”
CUDA GPU Roadmap

Jensen Huang’s Keynote at GTC 2010
ExaScale System Sketch
### ExaScale Chip Floorplan

<table>
<thead>
<tr>
<th>DRAM I/O</th>
<th>DRAM I/O</th>
<th>NW I/O</th>
<th>DRAM I/O</th>
<th>DRAM I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>SM</td>
<td>SM</td>
<td>SM</td>
<td>SM</td>
<td>SM</td>
</tr>
<tr>
<td>SM</td>
<td>SM</td>
<td>SM</td>
<td>SM</td>
<td>SM</td>
</tr>
<tr>
<td>SM</td>
<td>SM</td>
<td>SM</td>
<td>SM</td>
<td>SM</td>
</tr>
<tr>
<td>SM</td>
<td>SM</td>
<td>SM</td>
<td>SM</td>
<td>SM</td>
</tr>
</tbody>
</table>

**NOC**
- L2 Banks
- XBAR

- 17mm
- 10nm process
- 290mm²
Multiprocessors

• Multicore microprocessors
  – More than one processor per chip

• Requires explicitly parallel programming
  – Compare with instruction level parallelism
    • Hardware executes multiple instructions at once
    • Hidden from the programmer

  – Hard to do
    • Programming for performance
    • Load balancing
    • Optimizing communication and synchronization
SPEC CPU Benchmark

- Programs used to measure performance
  - Supposedly typical of actual workload
- Standard Performance Evaluation Corp (SPEC)
  - Develops benchmarks for CPU, I/O, Web, ...

- SPEC CPU2006
  - Elapsed time to execute a selection of programs
    - Negligible I/O, so focuses on CPU performance
  - Normalize relative to reference machine
  - Summarize as geometric mean of performance ratios
    - CINT2006 (integer) and CFP2006 (floating-point)

\[
\sqrt[n]{\prod_{i=1}^{n} \text{Execution time ratio}_i}
\]
SPEC Power Benchmark

- Power consumption of server at different workload levels
  - Performance: ssj_ops/sec
  - Power: Watts (Joules/sec)

\[
\text{Overall ssj\_ops per Watt} = \left( \sum_{i=0}^{10} \text{ssj\_ops}_i \right) \div \left( \sum_{i=0}^{10} \text{power}_i \right)
\]
### SPECpower_ssj2008 for X4

<table>
<thead>
<tr>
<th>Target Load %</th>
<th>Performance (ssj_ops/sec)</th>
<th>Average Power (Watts)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100%</td>
<td>231,867</td>
<td>295</td>
</tr>
<tr>
<td>90%</td>
<td>211,282</td>
<td>286</td>
</tr>
<tr>
<td>80%</td>
<td>185,803</td>
<td>275</td>
</tr>
<tr>
<td>70%</td>
<td>163,427</td>
<td>265</td>
</tr>
<tr>
<td>60%</td>
<td>140,160</td>
<td>256</td>
</tr>
<tr>
<td>50%</td>
<td>118,324</td>
<td>246</td>
</tr>
<tr>
<td>40%</td>
<td>920,35</td>
<td>233</td>
</tr>
<tr>
<td>30%</td>
<td>70,500</td>
<td>222</td>
</tr>
<tr>
<td>20%</td>
<td>47,126</td>
<td>206</td>
</tr>
<tr>
<td>10%</td>
<td>23,066</td>
<td>180</td>
</tr>
<tr>
<td>0%</td>
<td>0</td>
<td>141</td>
</tr>
<tr>
<td>Overall sum</td>
<td>1,283,590</td>
<td>2,605</td>
</tr>
<tr>
<td>(\Sigma)ssj_ops/ (\Sigma)power</td>
<td></td>
<td>493</td>
</tr>
</tbody>
</table>
Pitfall: Amdahl’s Law

- Improving an aspect of a computer and expecting a proportional improvement in overall performance

\[ T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{improvement factor}} + T_{\text{unaffected}} \]

- Example: multiply accounts for 80s/100s
  - How much improvement in multiply performance to get 5× overall?

  \[ 20 = \frac{80}{n} + 20 \]

- Can’t be done!

- Corollary: make the common case fast
Fallacy: Low Power at Idle

• Look back at X4 power benchmark
  – At 100% load: 295W
  – At 50% load: 246W (83%)
  – At 10% load: 180W (61%)

• Google data center
  – Mostly operates at 10% – 50% load
  – At 100% load less than 1% of the time

• Consider designing processors to make power proportional to load
Pitfall: MIPS as a Performance Metric

• MIPS: Millions of Instructions Per Second
  – Doesn’t account for
    • Differences in ISAs between computers
    • Differences in complexity between instructions

\[
\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6} = \frac{\text{Instruction count}}{\text{Instruction count} \times \text{CPI} \times 10^6} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}
\]

- CPI varies between programs on a given CPU
Chapter 2

• Not covering: procedure calling, linkers, object modules, java, arrays/pointers
Instruction Set

• The repertoire of instructions of a computer

• Different computers have different instruction sets
  – But with many aspects in common

• Early computers had very simple instruction sets
  – Simplified implementation

• Many modern computers also have simple instruction sets
The MIPS Instruction Set

• Used as the example throughout the book

• Stanford MIPS commercialized by MIPS Technologies (www.mips.com)

• Large share of embedded core market
  – Applications in consumer electronics, network/storage equipment, cameras, printers, ...

• Typical of many modern ISAs
  – See MIPS Reference Data tear-out card, and Appendixes B and E
Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination
  \[
  \text{add } a, b, c \; \# \; a \text{ gets } b + c
  \]

- All arithmetic operations have this form

- \textit{Design Principle 1: Simplicity favours regularity}
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost
Arithmetic Example

• C code:

\[ f = (g + h) - (i + j); \]

• Compiled MIPS code:

```
add t0, g, h   # temp t0 = g + h
add t1, i, j   # temp t1 = i + j
sub f, t0, t1  # f = t0 - t1
```
Register Operands

• Arithmetic instructions use register operands

• MIPS has a $32 \times 32$-bit register file
  – Use for frequently accessed data
  – Numbered 0 to 31
  – 32-bit data called a “word”

• Assembler names
  – $t0$, $t1$, ..., $t9$ for temporary values
  – $s0$, $s1$, ..., $s7$ for saved variables

• *Design Principle 2*: Smaller is faster
  – c.f. main memory: millions of locations
Register Operand Example

• **C code:**
  
  \[
  f = (g + h) - (i + j);
  \]
  
  – f, ..., j in $s0$, ..., $s4$

• **Compiled MIPS code:**

  ```
  add $t0, $s1, $s2
  add $t1, $s3, $s4
  sub $s0, $t0, $t1
  ```
Memory Operands

• Main memory used for composite data
  – Arrays, structures, dynamic data

• To apply arithmetic operations
  – Load values from memory into registers
  – Store result from register to memory

• Memory is byte addressed
  – Each address identifies an 8-bit byte

• Words are aligned in memory
  – Address must be a multiple of 4

• MIPS is Big Endian
  – Most-significant byte at least address of a word
  – *c.f.* Little Endian: least-significant byte at least address
Memory Operand Example 1

• C code:
  \[ g = h + A[8]; \]
  – g in $s1, h in $s2, base address of A in $s3

• Compiled MIPS code:
  – Index 8 requires offset of 32
    • 4 bytes per word
  \[
  \text{lw } \$t0, 32(\$s3) \quad \text{# load word}
  \text{add } \$s1, \$s2, \$t0
  \]
Memory Operand Example 2

• C code:
  
  \[
  \]
  
  – h in $s2, base address of A in $s3

• Compiled MIPS code:
  
  – Index 8 requires offset of 32

  \[
  \text{lw } \$t0, 32(\$s3) \quad \# \text{ load word}
  \]

  \[
  \text{add } \$t0, \$s2, \$t0
  \]

  \[
  \text{sw } \$t0, 48(\$s3) \quad \# \text{ store word}
  \]
Registers vs. Memory

• Registers are faster to access than memory

• Operating on memory data requires loads and stores
  – More instructions to be executed

• Compiler must use registers for variables as much as possible
  – Only spill to memory for less frequently used variables
  – Register optimization is important!
Immediate Operands

• Constant data specified in an instruction
  addi $s3, $s3, 4

• No subtract immediate instruction
  – Just use a negative constant
    addi $s2, $s1, -1

• Design Principle 3: Make the common case fast
  – Small constants are common
  – Immediate operand avoids a load instruction
The Constant Zero

• MIPS register 0 ($zero) is the constant 0
  – Cannot be overwritten

• Useful for common operations
  – E.g., move between registers
    add $t2, $s1, $zero
Unsigned Binary Integers

• Given an n-bit number

\[ x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_12^1 + x_02^0 \]

■ Range: 0 to \( +2^n - 1 \)

■ Example
  
  0000 0000 0000 0000 0000 0000 0000 1011\_2
  
  = 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0
  
  = 0 + \ldots + 8 + 0 + 2 + 1 = 11\_10

■ Using 32 bits
  
  0 to \( +4,294,967,295 \)
2s-Complement Signed Integers

- Given an n-bit number

\[ x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_12^1 + x_02^0 \]

- Range: \(-2^{n-1}\) to \(+2^{n-1} - 1\)

- Example

  1111 1111 1111 1111 1111 1111 1111 1100\(_2\)

  \[\begin{align*}
  &= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0 \\
  &= -2,147,483,648 + 2,147,483,644 = -4_{10}
  \end{align*}\]

- Using 32 bits

  - \(-2,147,483,648\) to \(+2,147,483,647\)
2s-Complement Signed Integers

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- \((-2^{n-1})\) can’t be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
  - 0: \(0000\ 0000\ ...\ 0000\)
  - \(-1\): \(1111\ 1111\ ...\ 1111\)
  - Most-negative: \(1000\ 0000\ ...\ 0000\)
  - Most-positive: \(0111\ 1111\ ...\ 1111\)
Signed Negation

• Complement and add 1
  – Complement means $1 \rightarrow 0$, $0 \rightarrow 1$

$$\overline{x + x} = 1111\ldots111_2 = -1$$

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

Example: negate $+2$

- $+2 = 0000\ 0000\ \ldots\ 0010_2$
- $-2 = 1111\ 1111\ \ldots\ 1101_2 + 1 = 1111\ 1111\ \ldots\ 1110_2$
Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- MIPS instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, ...
  - Regularity!
- Register numbers
  - $t0 – $t7 are reg’s 8 – 15
  - $t8 – $t9 are reg’s 24 – 25
  - $s0 – $s7 are reg’s 16 – 23
MIPS R-format Instructions

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

- **Instruction fields**
  - **op**: operation code (opcode)
  - **rs**: first source register register number
  - **rt**: second source register register number
  - **rd**: destination register register number
  - **shamt**: shift amount (000000 for now)
  - **funct**: function code (extends opcode)
R-format Example

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

**add $t0, $s1, $s2**

<table>
<thead>
<tr>
<th>special</th>
<th>$s1</th>
<th>$s2</th>
<th>$t0</th>
<th>0</th>
<th>add</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17</td>
<td>18</td>
<td>8</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>000000</td>
<td>10001</td>
<td>10010</td>
<td>01000</td>
<td>00000</td>
<td>100000</td>
</tr>
</tbody>
</table>

0000000100011001001000000000100000_2 = 02324020_{16}
MIPS I-format Instructions

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>constant or address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: $-2^{15}$ to $+2^{15} - 1$
  - Address: offset added to base address in rs

- **Design Principle 4**: Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible
Conditional Operations

- Branch to a labeled instruction if a condition is true
  – Otherwise, continue sequentially

- `beq rs, rt, L1`
  – if (rs == rt) branch to instruction labeled L1;

- `bne rs, rt, L1`
  – if (rs != rt) branch to instruction labeled L1;

- `j L1`
  – unconditional jump to instruction labeled L1
Compiling If Statements

- **C code:**
  
  ```c
  if (i==j) f = g+h;
  else f = g-h;
  ```

  - f, g, ... in $s0, $s1, ...

- **Compiled MIPS code:**

  ```
  bne $s3, $s4, Else
  add $s0, $s1, $s2
  j Exit
  Else: sub $s0, $s1, $s2
  Exit: ...
  ```

  Assembler calculates addresses
Compiling Loop Statements

• C code:
  
  ```
  while (save[i] == k) i += 1;
  ```
  
  — i in $s3, k in $s5, address of save in $s6

• Compiled MIPS code:
  
  ```
  Loop: sll $t1, $s3, 2
       add $t1, $t1, $s6
       lw $t0, 0($t1)
       bne $t0, $s5, Exit
       addi $s3, $s3, 1
       j Loop

  Exit: ...
  ```
More Conditional Operations

• Set result to 1 if a condition is true
  – Otherwise, set to 0

• `slt rd, rs, rt`
  – if (rs < rt) rd = 1; else rd = 0;

• `slti rt, rs, constant`
  – if (rs < constant) rt = 1; else rt = 0;

• Use in combination with `beq`, `bne`

  `slt $t0, $s1, $s2`  # if ($s1 < $s2)
  `bne $t0, $zero, L`  # branch to L
Branch Instruction Design

• Why not b\texttt{lt}, b\texttt{ge}, etc?
• Hardware for $<$, $\geq$, ... slower than $=$, $\neq$
  – Combining with branch involves more work per instruction, requiring a slower clock
  – All instructions penalized!
• \texttt{beq} and \texttt{bne} are the common case
• This is a good design compromise
Register Usage

• $a0 – $a3: arguments (reg’s 4 – 7)
• $v0, $v1: result values (reg’s 2 and 3)
• $t0 – $t9: temporaries
  – Can be overwritten by callee
• $s0 – $s7: saved
  – Must be saved/restored by callee
• $gp: global pointer for static data (reg 28)
• $sp: stack pointer (reg 29)
• $fp: frame pointer (reg 30)
• $ra: return address (reg 31)
Procedure Call Instructions

• Procedure call: jump and link
  
  jal ProcedureLabel
  – Address of following instruction put in $ra
  – Jumps to target address

• Procedure return: jump register
  
  jr $ra
  – Copies $ra to program counter
  – Can also be used for computed jumps
    • e.g., for case/switch statements
Memory Layout

- **Text**: program code
- **Static data**: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp$ initialized to address allowing ±offsets into this segment
- **Dynamic data**: heap
  - E.g., malloc in C, new in Java
- **Stack**: automatic storage
32-bit Constants

• Most constants are small
  – 16-bit immediate is sufficient

• For the occasional 32-bit constant
  
  lui rt, constant
  – Copies 16-bit constant to left 16 bits of rt
  – Clears right 16 bits of rt to 0

lhi $s0, 61

ori $s0, $s0, 2304
Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>constant or address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- PC-relative addressing
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time
Jump Addressing

• Jump (j and jal) targets could be anywhere in text segment
  – Encode full address in instruction

<table>
<thead>
<tr>
<th>op</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>

(Pseudo)Direct jump addressing

  - Target address = PC_{31...28} : (address \times 4)
Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code

- Example
  
  ```
  beq $s0,$s1, L1
  ↓
  bne $s0,$s1, L2
  j L1
  
  L2: ...
  ```
Addressing Mode Summary

1. Immediate addressing

   \[
   \begin{array}{ccc}
   \text{op} & \text{rs} & \text{rt} \\
   & \text{Immediate} & \\
   \end{array}
   \]

2. Register addressing

   \[
   \begin{array}{cccc}
   \text{op} & \text{rs} & \text{rt} & \text{rd} \\
   & \text{... funct} & & \\
   \end{array}
   \]

3. Base addressing

   \[
   \begin{array}{ccc}
   \text{op} & \text{rs} & \text{rt} \\
   & \text{Address} & \\
   \end{array}
   \]

4. PC-relative addressing

   \[
   \begin{array}{ccc}
   \text{op} & \text{rs} & \text{rt} \\
   & \text{Address} & \\
   \end{array}
   \]

5. Pseudodirect addressing

   \[
   \begin{array}{c}
   \text{op} \\
   \text{Address} \\
   \end{array}
   \]

Chapter 2 — Instructions:
Language of the Computer — 74
Synchronization

• Two processors sharing an area of memory
  – P1 writes, then P2 reads
  – Data race if P1 and P2 don’t synchronize
    • Result depends on order of accesses

• Hardware support required
  – Atomic read/write memory operation
  – No other access to the location allowed between the read and write

• Could be a single instruction
  – E.g., atomic swap of register ↔ memory
  – Or an atomic pair of instructions
Synchronization in MIPS

- **Load linked:** `ll rt, offset(rs)`

- **Store conditional:** `sc rt, offset(rs)`
  - Succeeds if location not changed since the `ll`
    - Returns 1 in `rt`
  - Fails if location is changed
    - Returns 0 in `rt`

- **Example: atomic swap (to test/set lock variable)**

```assembly
try: add $t0,$zero,$s4 ; copy exchange value
   ll $t1,0($s1)    ; load linked
   sc $t0,0($s1)    ; store conditional
   beq $t0,$zero,try ; branch store fails
   add $s4,$zero,$t1 ; put load value in $s4
```
ARM & MIPS Similarities

- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

<table>
<thead>
<tr>
<th></th>
<th>ARM</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date announced</td>
<td>1985</td>
<td>1985</td>
</tr>
<tr>
<td>Instruction size</td>
<td>32 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td>Address space</td>
<td>32-bit flat</td>
<td>32-bit flat</td>
</tr>
<tr>
<td>Data alignment</td>
<td>Aligned</td>
<td>Aligned</td>
</tr>
<tr>
<td>Data addressing modes</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>Registers</td>
<td>15 × 32-bit</td>
<td>31 × 32-bit</td>
</tr>
<tr>
<td>Input/output</td>
<td>Memory mapped</td>
<td>Memory mapped</td>
</tr>
</tbody>
</table>
Compare and Branch in ARM

• Uses condition codes for result of an arithmetic/logical instruction
  – Negative, zero, carry, overflow
  – Compare instructions to set condition codes without keeping the result

• Each instruction can be conditional
  – Top 4 bits of instruction word: condition value
  – Can avoid branches over single instructions
Instruction Encoding

Chapter 2 — Instructions: Language of the Computer — 79
The Intel x86 ISA

• Evolution with backward compatibility
  – 8080 (1974): 8-bit microprocessor
    • Accumulator, plus 3 index-register pairs
  – 8086 (1978): 16-bit extension to 8080
    • Complex instruction set (CISC)
  – 8087 (1980): floating-point coprocessor
    • Adds FP instructions and register stack
  – 80286 (1982): 24-bit addresses, MMU
    • Segmented memory mapping and protection
    • Additional addressing modes and operations
    • Paged memory mapping as well as segments
The Intel x86 ISA

• Further evolution...
  – i486 (1989): pipelined, on-chip caches and FPU
    • Compatible competitors: AMD, Cyrix, ...
  – Pentium (1993): superscalar, 64-bit datapath
    • Later versions added MMX (Multi-Media eXtension) instructions
    • The infamous FDIV bug
    • New microarchitecture (see Colwell, *The Pentium Chronicles*)
  – Pentium III (1999)
    • Added SSE (Streaming SIMD Extensions) and associated registers
  – Pentium 4 (2001)
    • New microarchitecture
    • Added SSE2 instructions
The Intel x86 ISA

- And further...
  - **AMD64 (2003):** extended architecture to 64 bits
  - **EM64T – Extended Memory 64 Technology (2004):**
    - AMD64 adopted by Intel (with refinements)
    - Added SSE3 instructions
  - **Intel Core (2006):**
    - Added SSE4 instructions, virtual machine support
  - **AMD64 (announced 2007): SSE5 instructions**
    - Intel declined to follow, instead...
  - **Advanced Vector Extension (announced 2008):**
    - Longer SSE registers, more instructions
- If Intel didn’t extend with compatibility, its competitors would!
  - Technical elegance ≠ market success
x86 Instruction Encoding

- Variable length encoding
  - Postfix bytes specify addressing mode
  - Prefix bytes modify operation
- Operand length, repetition, locking, ...
Fallacies

• Powerful instruction $\Rightarrow$ higher performance
  – Fewer instructions required
  – But complex instructions are hard to implement
    • May slow down all instructions, including simple ones
  – Compilers are good at making fast code from simple instructions

• Use assembly code for high performance
  – But modern compilers are better at dealing with modern processors
  – More lines of code $\Rightarrow$ more errors and less productivity
Fallacies

• Backward compatibility $\Rightarrow$ instruction set doesn’t change
  — But they do accrete more instructions
Concluding Remarks

• Measure MIPS instruction executions in benchmark programs
  – Consider making the common case fast
  – Consider compromises

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>MIPS examples</th>
<th>SPEC2006 Int</th>
<th>SPEC2006 FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add, sub, addi</td>
<td>16%</td>
<td>48%</td>
</tr>
<tr>
<td>Data transfer</td>
<td>lw, sw, lb, lbu, lh, lhu, sb, lui</td>
<td>35%</td>
<td>36%</td>
</tr>
<tr>
<td>Logical</td>
<td>and, or, nor, andi, ori, sll, srl</td>
<td>12%</td>
<td>4%</td>
</tr>
<tr>
<td>Cond. Branch</td>
<td>beq, bne, slt, slti, sltiu</td>
<td>34%</td>
<td>8%</td>
</tr>
<tr>
<td>Jump</td>
<td>j, jr, jal</td>
<td>2%</td>
<td>0%</td>
</tr>
</tbody>
</table>
Appendix A: Instructions
Logical Operations

- Instructions for bitwise manipulation

<table>
<thead>
<tr>
<th>Operation</th>
<th>C</th>
<th>Java</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift left</td>
<td>&lt;&lt;</td>
<td>&lt;&lt;</td>
<td>sll</td>
</tr>
<tr>
<td>Shift right</td>
<td>&gt;&gt;</td>
<td>&gt;&gt;&gt;</td>
<td>srl</td>
</tr>
<tr>
<td>Bitwise AND</td>
<td>&amp;</td>
<td>&amp;</td>
<td>and, andi</td>
</tr>
<tr>
<td>Bitwise OR</td>
<td></td>
<td></td>
<td>or, ori</td>
</tr>
<tr>
<td>Bitwise NOT</td>
<td>~</td>
<td>~</td>
<td>nor</td>
</tr>
</tbody>
</table>

- Useful for extracting and inserting groups of bits in a word
Shift Operations

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

- **shamt**: how many positions to shift
- **Shift left logical**
  - Shift left and fill with 0 bits
  - `sll` by `i` bits multiplies by `2^i`
- **Shift right logical**
  - Shift right and fill with 0 bits
  - `srl` by `i` bits divides by `2^i` (unsigned only)
AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

and $t0, $t1, $t2

<table>
<thead>
<tr>
<th></th>
<th>0000 0000 0000 0000 0000 0000 1101 1100 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t2</td>
<td></td>
</tr>
<tr>
<td>$t1</td>
<td>0000 0000 0000 0000 0000 0011 1100 0000 0000</td>
</tr>
<tr>
<td>$t0</td>
<td>0000 0000 0000 0000 0000 0000 1100 0000 0000</td>
</tr>
</tbody>
</table>
## OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

```
or $t0, $t1, $t2
```

<table>
<thead>
<tr>
<th>$t2</th>
<th>0000 0000 0000 0000 0000 1101 1100 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t1</td>
<td>0000 0000 0000 0000 0011 1100 0000 0000</td>
</tr>
<tr>
<td>$t0</td>
<td>0000 0000 0000 0000 0011 1101 1100 0000</td>
</tr>
</tbody>
</table>
NOT Operations

• Useful to invert bits in a word
  – Change 0 to 1, and 1 to 0

• MIPS has NOR 3-operand instruction
  – a NOR b == NOT ( a OR b )

\[
\text{nor } \$t0, \$t1, \$zero
\]

Register 0: always read as zero

\[
\begin{array}{c}
\$t1 \\
0000 0000 0000 0000 0011 1100 0000 0000
\end{array}
\]

\[
\begin{array}{c}
\$t0 \\
1111 1111 1111 1111 1100 0011 1111 1111
\end{array}
\]