Review: MIPS Addressing Modes/Instruction Formats

R-format:
Register (direct) 6 5 5 5 5 6  
\[ \text{op} \quad \text{rs} \quad \text{rt} \quad \text{rd} \quad \text{sht} \quad \text{fun} \]

J-format:
I-format:
Immediate  
\[ \text{op} \quad \text{rs} \quad \text{rt} \quad \text{immed} \]

Base+offset displacement 6 5 5 16  
\[ \text{op} \quad \text{rs} \quad \text{rt} \quad \text{immed} \]

PC-relative  
\[ \text{op} \quad \text{rs} \quad \text{rt} \quad \text{immed} \]

J-format: 6 26  
\[ \text{op} \quad \text{addr.} \]

Memory

CS420/520 Lec 4 ISA.2  UC. Colorado Springs  Adapted from ©UCB97 & ©UCB03
An instruction is a binary code, which specifies a basic operation (e.g. add, subtract, and, or) for the computer.

- **Operation Code**: defines the operation type
- **Operands**: operation source and destination

### Basic Issues in Instruction Set Design

--- What operations (and how many) should be provided

LD/ST/INC/BRN sufficient to encode any computation

But not useful because programs too long!

--- How (and how many) operands are specified

Most operations are dyadic (e.g., \( A := B + C \))

Some are monadic  (e.g., \( A := \neg B \))

--- How to encode these into consistent instruction formats

Instructions should be multiples of basic data/address widths

**Typical instruction set:**

- 32 bit word
- basic operand addresses are 32 bits long
- basic operands, like integers, are 32 bits long
- in general case, instruction could reference 3 operands (\( A := B + C \))

challenge: encode operations in a small number of bits!
Execution Cycle

- **Instruction Fetch**
  - Obtain instruction from program storage

- **Instruction Decode**
  - Determine required actions and instruction size

- **Operand Fetch**
  - Locate and obtain operand data

- **Execute**
  - Compute result value or status

- **Result Store**
  - Deposit results in storage for later use

- **Next Instruction**
  - Determine successor instruction

Parallelism is important!

What Must be Specified?

- **Instruction Fetch**
  - Instruction Format or Encoding
    - how is it decoded?
  - Location of operands and result
    - where other than memory?
    - how many explicit operands?
    - how are memory operands located?
    - which can or cannot be in memory?

- **Operand Fetch**

- **Execute**

- **Result Store**

- **Next Instruction**

- Data type and Size

- Operations
  - what are supported

- Successor instruction
  - jumps, conditions, branches

- *fetch-decode-execute is implicit!
Basic ISA Classes

Accumulator: (earliest machines)
- 1 address
  - load/store A
    - acc ← mem[A]
  - add A
    - acc ← acc + mem[A]

Stack: (HP calculator, Java virtual machines)
- 0 address
  - add
    - tos ← tos + next

Register (register-Memory): (e.g. Intel 80x86, Motorola 68xxx)
- 2 address
  - add A B
    - EA(A) ← EA(A) + EA(B)
- 3 address
  - add A B C
    - EA(A) ← EA(B) + EA(C)

Register (Load/Store): (e.g. SPARC, MIPS, PowerPC)
- 3 address
  - add Ra Rb Rc
    - Ra ← Rb + Rc
  - load Ra Rb
    - Ra ← mem[Rb]
  - store Ra Rb
    - mem[Rb] ← Ra

Memory – to - memory: no more shipping today

Classifying ISAs

1. Dimension 1: Where other than memory?
   - Accumulator
   - Stack
   - A set of registers

2. Naming Implicitly or explicitly?
   - Implicitly:
     - Accumulator
     - Stack; operands identified by TOS
   - Explicitly
     - General-purpose register architectures
       - Either registers or memory locations
Comparing Instructions

Comparing Number of Instructions

- Code sequence for $C = A + B$ for four classes of instruction sets:
  
  \[
  \begin{array}{cccc}
  \text{Stack} & \text{Accumulator} & \text{Register} & \text{Register (register-memory)} \\
  \text{Push A} & \text{Load A} & \text{Load R1,A} & \text{Load R1,A} \\
  \text{Push B} & \text{Add B} & \text{Add R1,B} & \text{Load R2,B} \\
  \text{Add} & \text{Store C} & \text{Store C,R1} & \text{Add R3,R1,R2} \\
  \text{Pop C} & \text{--} & \text{--} & \text{Store C,R3} \\
  \end{array}
  \]

  \[
  \begin{align*}
  S[\text{tos} - 4] &= S[\text{tos}] \text{ op } S[\text{tos} - 4]; \\
  \text{tos} &= \text{tos} - 4; \\
  \end{align*}
  \]

  Number of Instructions? Cycles per instruction?
**General Purpose Registers Dominate**

Since 1975 all machines use general purpose registers
° (Java Virtual Machine adopts Stack architecture)
° Advantages of registers
  • registers are faster than memory
  • registers are easier for a compiler to use
    - e.g., \((A*B) - (C*D) - (E*F)\) can do multiplies in any order
      vs. stack \((S[tos-4] = S[tos] \text{ op } S[tos-4]; \text{ tos = tos} - 4)\)
  • registers can hold variables
    - memory traffic is reduced, so program is sped up
      (since registers are faster than memory)
    - code density improves (since register named with fewer bits
      than memory location)
  • registers are efficient in pipelining
° how many registers are sufficient?
  • Compilers reserve some for expression evaluation, parameter
    passing, and the remainder to hold variables.

---

**Examples of Register Usage**

<table>
<thead>
<tr>
<th>Type of architecture</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-S</td>
<td>Alpha, ARM, SPARC, MIPS, Power PC. TM32</td>
</tr>
<tr>
<td>R-M</td>
<td>IBM 360/370, Intel 80x86, Motorola 68000</td>
</tr>
<tr>
<td>M-M</td>
<td>VAX (also has 3-operand formats)</td>
</tr>
<tr>
<td>M-M</td>
<td>VAX (also has 2-operand formats)</td>
</tr>
</tbody>
</table>

Typical combinations of memory operands and total operands per instruction
Example:

In VAX:  
**ADDL (R9), (R10), (R11)**  
\[
\text{mem}[R9] \leftarrow \text{mem}[R10] + \text{mem}[R11]
\]

VAX: richest of addressing modes  
fewest restrictions on memory addressing

In MIPS:  
\[
\begin{align*}
\text{lw} & \quad \text{R1, (R10)}; \quad \text{load a word} \\
\text{lw} & \quad \text{R2, (R11)} \\
\text{add} & \quad \text{R3, R1, R2}; \quad \text{R3} \leftarrow \text{R1} + \text{R2} \\
\text{sw} & \quad \text{R3, (R9)}; \quad \text{store a word}
\end{align*}
\]

Pros and Cons of Number Memory Operands/Operands

° Register–register: 0 memory operands/instr, 3 (register) operands/instr  
  + Simple, fixed-length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute  
  – Higher instruction count than architectures with memory references in instructions. Some instructions are short and bit encoding may be wasteful.

° Register–memory (1,2)  
  + Data can be accessed without loading first. Instruction format tends to be easy to encode and yields good density.  
  – Operands are not equivalent since a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction varies by operand location.

° Memory–memory (2,2) or (3,3)  
  + Most compact. Doesn’t waste registers for temporaries.  
  – Large variation in instruction size, especially for three-operand instructions. Also, large variation in work per instruction. Memory accesses create memory bottleneck.
**Memory Addressing**

Since 1980 almost every machine uses addresses to level of 8-bits (byte)

2 questions for design of ISA:
- Since could read a 32-bit word as four loads of bytes from sequential byte addresses or as one load word from a single byte address,

  How do byte addresses map onto words?

  Can a word be placed on any byte boundary?

**Alignment Restriction**

Alignment issue:
- access objects larger than a byte must be aligned

  Alignment: require that objects fall on address that is multiple of their size

  An access to an object of size $s$ bytes at byte address $A$ is aligned if $A \mod s = 0$

  Alignment leads to faster data transfers

  Example:
**Addressing Objects**

BigEndian: address of most significant (MSB) = **word address**
IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

LittleEndian: address of least significant (LSB) = **word address**
Intel 80x86, DEC Vax

**BIG Endian versus Little Endian**

Example 1: Memory layout of a number #ABCDEFGH:
SW $4(#ABCDEFGH), 1000($0)

In Big Endian:

<table>
<thead>
<tr>
<th>GH</th>
<th>EF</th>
<th>CD</th>
<th>AB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1003</td>
<td>1002</td>
<td>1001</td>
<td>1000</td>
</tr>
</tbody>
</table>

In Little Endian:

<table>
<thead>
<tr>
<th>GH</th>
<th>EF</th>
<th>CD</th>
<th>AB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>1000</td>
<td>1002</td>
<td>1003</td>
</tr>
</tbody>
</table>

Example 2: Memory layout of a number #FF00H

How about load?
## Addressing Modes

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Add R4,R3</td>
<td>$R4 \leftarrow R4 + R3$</td>
</tr>
<tr>
<td>Immediate</td>
<td>Add R4,#3</td>
<td>$R4 \leftarrow R4 + 3$</td>
</tr>
<tr>
<td>Displacement</td>
<td>Add R4,100(R1)</td>
<td>$R4 \leftarrow R4 + Mem[100+R1]$</td>
</tr>
<tr>
<td>Register indirect</td>
<td>Add R4,(R1)</td>
<td>$R4 \leftarrow R4 + Mem[R1]$</td>
</tr>
<tr>
<td>Indexed</td>
<td>Add R3,(R1+R2)</td>
<td>$R3 \leftarrow R3 + Mem[R1+R2]$</td>
</tr>
<tr>
<td>Direct or absolute</td>
<td>Add R1,(1001)</td>
<td>$R1 \leftarrow R1 + Mem[1001]$</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>Add R1,@(R3)</td>
<td>$R1 \leftarrow R1 + Mem[Mem[R3]]$</td>
</tr>
<tr>
<td>Auto-increment</td>
<td>Add R1,(R2)+</td>
<td>$R1 \leftarrow R1 + Mem[R2]; R2 \leftarrow R2+d$</td>
</tr>
<tr>
<td>Auto-decrement</td>
<td>Add R1,–(R2)</td>
<td>$R2 \leftarrow R2–d; R1 \leftarrow R1 + Mem[R2]$</td>
</tr>
<tr>
<td>Scaled</td>
<td>Add R1,100(R2)[R3]</td>
<td>$R1 \leftarrow R1 + Mem[100+R2+R3*d]$</td>
</tr>
</tbody>
</table>

## Addressing Mode Illustrations

- (a) Immediate
- (b) Direct
- (c) Indirect
- (d) Register
- (e) Memory
- (f) Displacement
- (g) Stack
- (h) Top-of-Stack Register
Addressing Mode:

- Addressing modes have the ability to significantly reduce instruction counts.
- They also add to the complexity of building a machine, and may increase the CPI of computers.
- The usage of various addressing modes is important in helping the architect choose what to include.

Addressing Mode Usage (VAX)

What it tells?

- Important addressing modes: Displacement, Immediate, Register Indirect.
Displacement Address Size (Alpha)

- Values are widely distributed
- X-axis is in power of 2
- 1% of address size > 16-bits

What it tells?
- displacement size should be 12-16 bits, capturing 75%-99%

Why important?
- directly affects the instruction length

Immediate Frequency (Alpha)

What it tells?
- About one quarter of data transfers and ALU operations have an immediate Operand.
The distribution of Immediate Size (Alpha)

- Values are widely distributed
- X-axis is in power of 2
- 20% of size > 16-bits

What it tells?
- Immediate size should be 8-16 bits, capturing 50%-80%

Why important?
- Affects the instruction length too

Summary of Addressing Modes

- Data Addressing modes that are important:
  Displacement, Immediate, Register Indirect
- Displacement size should be 12 to 16 bits
- Immediate size should be 8 to 16 bits
### Data Types and Sizes

**Bit:** 0, 1

- **Bit String:** sequence of bits of a particular length
  - 4 bits is a nibble
  - 8 bits is a byte
  - 16 bits is a half-word
  - 32 bits is a word

**Character:**
- ASCII 7 bit code
- EBCDIC 8 bit code (IBM)
- UNICODE 16 bit code (Java)

**Decimal:**
- digits 0-9 encoded as 0000b thru 1001b
- two decimal digits packed per 8 bit byte

**Integers:**
- **Sign & Magnitude:** 0X vs. 1X
- 1’s Complement: 0X vs. 1(~X)
- 2’s Complement: 0X vs. (1’s comp) + 1
  - Positive #’s same in all
  - First 2 have two zeros
  - Last one usually chosen

**Floating Point:**
- Single Precision
- Double Precision
- Extended Precision
  - How many +/- sign’s?
  - Where is decimal pt?
  - How are +/- exponents represented?

\[ M \times R^E \]

\[ \text{mantissa} \]

\[ \text{base} \]

- **Operand Size Usage (on 64-bit addresses):**

<table>
<thead>
<tr>
<th>Type</th>
<th>Floating-point average</th>
<th>Integer average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double word</td>
<td>70%</td>
<td>59%</td>
</tr>
<tr>
<td>(64 bits)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Word</td>
<td>29%</td>
<td>25%</td>
</tr>
<tr>
<td>(32 bits)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Half word</td>
<td>5%</td>
<td>5%</td>
</tr>
<tr>
<td>(16 bits)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Byte</td>
<td>1%</td>
<td>10%</td>
</tr>
<tr>
<td>(8 bits)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Support these data sizes and types:
  - 8-bit, 16-bit, 32-bit integers and
  - 32-bit and 64-bit IEEE 754 floating point numbers
Typical Operations

Data Movement
- Load (from memory)
- Store (to memory)
- Memory-to-memory move
- Register-to-register move
- Input (from I/O device)
- Output (to I/O device)
- Push, pop (to/from stack)

Arithmetic
- Integer (binary + decimal) or FP
- Add, Subtract, Multiply, Divide

Logical
- Not, and, or, set, clear

Shift
- Shift left/right, rotate left/right

Control (Jump/Branch)
- Unconditional, conditional

Subroutine Linkage
- Call, return

Interrupt
- Trap, return

Synchronization
- Test & set (atomic r-m-w)

String
- Move, compare, search, translate

Top 10 80x86 Instructions

<table>
<thead>
<tr>
<th>Rank</th>
<th>Instruction</th>
<th>Integer</th>
<th>Average</th>
<th>Percent</th>
<th>Total executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load</td>
<td>22%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>conditional branch</td>
<td>20%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>compare</td>
<td>16%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>store</td>
<td>12%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>add</td>
<td>8%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>and</td>
<td>6%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>sub</td>
<td>5%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>move register-register</td>
<td>4%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>call</td>
<td>1%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>return</td>
<td>1%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Simple instructions dominate instruction frequency
**Most Popular MIPS Instructions**

- **Left:** SPECint2000 (96%)
- **Right:** SPECfp2000 (97%)

**Operation Summary**

- Support these simple instructions, since they will dominate the number of instructions executed:
  - load,
  - store,
  - add,
  - subtract,
  - move register-register,
  - and,
  - shift,
  - compare equal, compare not equal,
  - branch (with a PC-relative address at least 8-bits long),
  - jump,
  - call,
  - return;
Instructions for Control Flow (Alpha)

- Conditional branches; jumps; procedure call/return

Branch Distances (Alpha)

- Branch distances in terms of number of instructions
  Between the target and the branch instruction.

What it tells?

- Most branches are to targets that can be encoded in 4-8 bits; short
  displacement fields often suffice for branches;
  Important if coding density is the issue!
### Conditional Branch Options

<table>
<thead>
<tr>
<th>Name</th>
<th>Examples</th>
<th>How condition is tested</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cond</td>
<td>x86, ARM, PowerPC, SPARC, SuperH</td>
<td>Tests special bits set by ALU operations, possibly under program control.</td>
<td>Sometimes condition is set for free.</td>
<td>CC is extra state. Condition codes constrain the ordering of instructions since they pass information from one instruction to a branch.</td>
</tr>
<tr>
<td>Cond pr Int</td>
<td>Alpha, MIPS</td>
<td>Tests arbitrary register with the result of a comparison.</td>
<td>Simple.</td>
<td>Uses up a register.</td>
</tr>
<tr>
<td>Cond pr Br</td>
<td>PA-RISC, VAX</td>
<td>Compare is part of the branch. Often compare is limited to subset.</td>
<td>One instruction rather than two for a branch.</td>
<td>May be too much work per instruction for pipelined execution.</td>
</tr>
</tbody>
</table>

### Frequency of Compares (Alpha)

- **Not equal**: 5% (Floating-point average: 2% Integer average: 18%)
- **Equal**: 19% (Floating-point average: 19% Integer average: 18%)
- **Greater than or equal**: 11% (Floating-point average: 11% Integer average: 18%)
- **Greater than**: 0% (Floating-point average: 0% Integer average: 0%)
- **Less than or equal**: 33% (Floating-point average: 33% Integer average: 44%)
- **Less than**: 34% (Floating-point average: 34% Integer average: 35%)

**What it tells?**

Less than (or equal) dominates
Encoding an Instruction Set

• Balance several competing forces
  • The desire to have as many registers and addressing modes
  • The impact of the size of the registers and addressing mode fields on the average instruction size and hence the average program size
  • A desire to have instructions encoded into lengths that will be easy to handle in a pipelined implementation; many desktop and server architects have chosen to use a fixed-length instruction to gain implementation benefits while sacrificing average code size

Example: add EAX, 1000 (EBX)

\[ 1 + 1 + 4 = 6 \text{ bytes (in 80x86 32-bit mode)} \]

Generic Examples of Instruction Formats

<table>
<thead>
<tr>
<th>Operation and no. of operands</th>
<th>Address specifier 1</th>
<th>Address field 1</th>
<th>Address specifier 2</th>
<th>Address field 2</th>
<th>Address specifier 3</th>
<th>Address field 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Variable (e.g., VAX, Intel 80x86)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>low avg. code size -- best when many addressing modes but poor perf.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address specifier 1</th>
<th>Address field 1</th>
<th>Address specifier 2</th>
<th>Address field 2</th>
<th>Address specifier 3</th>
<th>Address field 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>easy decoding for compiler, easy pipelining but wasted bits in instr.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address specifier 1</th>
<th>Address field 1</th>
<th>Address specifier 2</th>
<th>Address field 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c) Hybrid (e.g., IBM 360/70, MIPS16, Thumb, TI TMS320C54x)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tradeoff: multiple of bytes, instead of arbitrary bit length</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Instruction Formats and Reduced Code Size

- If code size is most important, use variable length instructions
- If performance is most important, use fixed length instructions – ease of decoding
- Recent embedded machines (ARM, MIPS) added optional mode to execute subset of 16-bit wide instructions (Thumb, MIPS16), which both claim a code size reduction of up to 40%
- Some architectures actually exploring on-the-fly hardware decompression for more density (IBM).

Instruction Set Architectures

- Class of ISA
- Memory addressing
- Addressing modes
- Types and sizes of operands
- Operations
- Control flow instructions
- Encoding an ISA
Reduced Instruction Set Computer (RISC)

Key elements

- A large number of general-purpose registers, and/or the use of compiler technology to optimize register usage
- A limited and simple instruction set
- An emphasis on optimizing the instruction set
  - A single instruction size (typically 4 bytes)
  - Register-to-register operations
    » No operations that combines load/store with arithmetic
  - A small number of data addressing modes
  - Simple addressing modes
    » No indirect addressing
  - Simple instruction formats

CISC and RISC Characteristics

Characteristics of Some CISCs, RISCs, and Superscalar Processors

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Complex Instruction Set (CISC/Computer)</th>
<th>Reduced Instruction Set (RISC) Computer</th>
<th>Superscalar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year developed</td>
<td>IBM 370/168</td>
<td>SPARC</td>
<td>PowerPC</td>
</tr>
<tr>
<td></td>
<td>VAX 11/789</td>
<td>MIPS R4000</td>
<td>Ultra SPARC</td>
</tr>
<tr>
<td></td>
<td>Inst. 8046</td>
<td>MIPS R10000</td>
<td>MIPS R10000</td>
</tr>
<tr>
<td>Number of instructions</td>
<td>208</td>
<td>1997</td>
<td>1993</td>
</tr>
<tr>
<td></td>
<td>1978</td>
<td>1991</td>
<td>1993</td>
</tr>
<tr>
<td></td>
<td>1989</td>
<td></td>
<td>1996</td>
</tr>
<tr>
<td>Instruction size (bytes)</td>
<td>2–6</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>2–57</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>1–11</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Addressing modes</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>22</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Number of general-purpose registers</td>
<td>16</td>
<td>40–520</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Control memory size (Kbits)</td>
<td>420</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>480</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>246</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Cache size (KBytes)</td>
<td>04</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>04</td>
<td>128</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>10–32</td>
<td>64</td>
</tr>
</tbody>
</table>
CISC Program size vs. RISC program size

• Is it certain that a CISC program will be smaller than a corresponding RISC program?
  
  • in many cases, a CISC program, expressed in symbolic machine language, may be shorter (fewer instructions)
  
  • but the number of bits of memory occupied may not be noticeably smaller

Machine Examples: Address & Registers

<table>
<thead>
<tr>
<th></th>
<th>2^{10} x 8 bit bytes</th>
<th>acc, index, count</th>
<th>stack, string</th>
<th>code, stack, data segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel 8086</td>
<td>AX, BX, CX, DX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SP, BP, SI, DI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CS, SS, DS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IP, Flags</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VAX 11</td>
<td>2^{32} x 8 bit bytes</td>
<td>r15-- program counter</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16 x 32 bit GPRs</td>
<td>r14-- stack pointer</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>13 2 bit PC</td>
<td>r13-- frame pointer</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>r12-- argument ptr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MC 68000</td>
<td>2^{24} x 8 bit bytes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 x 32 bit GPRs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7 x 32 bit addr reg</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 x 32 bit SP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 x 32 bit PC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIPS</td>
<td>2^{32} x 8 bit bytes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32 x 32 bit GPRs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32 x 32 bit FPRs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>HI, LO, PC</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Concluding Remarks

• Changes in 1990s
  – Address size doubles
    » Instruction sets: 32-bit addresses → 64-bit addresses
    » Registers: 32-bit → 64-bit
  – Optimization of cache performance
    » Pre-fetch instructions were added (Memory Hierarchy)
  – Support for Multimedia
    » Instruction sets extended for MM and DSP applications

• Trends in ISA design
  – Long instruction words
    » More instruction-level parallelism (Pipelining)
  – Blending general-purpose and DSP architectures
  – 80x86 emulation
    » Given the popularity of software for 80x86 architecture, see if changes to the instruction sets can improve performance, cost or power when emulating the 80x86 architecture

Lecture Summary: ISA

° Use general purpose registers with a load-store architecture;
° Support these addressing modes: displacement (with an address offset size of 12 to 16 bits), immediate (size 8 to 16 bits), and register deferred;
° Support these simple instructions, since they will dominate the number of instructions executed: load, store, add, subtract, move register-register, and, shift, compare equal, compare not equal, branch (with a PC-relative address at least 8-bits long), jump, call, and return;
° Support these data sizes and types: 8-bit, 16-bit, 32-bit integers and 64-bit IEEE 754 floating point numbers;
° Use fixed instruction encoding if interested in performance and use variable instruction encoding if interested in code size;
° Provide at least 16 general purpose registers plus separate floating-point registers, be sure all addressing modes apply to all data transfer instructions, and aim for a minimalist instruction set.
Reading

- Reading:
  CO4: Chapter 2 (MIPS)
  CA 5: Appendix A (ISA)

- Preview:
  CO4: Chapter 4 (The Processor)

Links to Information Assurance related Websites

- ITU (International Telecommunication Union): http://www.itu.int/
- Internet Society (ISOC): http://www.isoc.org/
- The Internet Engineering Task Force (IETF): http://www.ietf.org/
- Internet Architecture Board (IAB): http://www.iab.org/
- International Organization for Standardization (ISO): http://www.iso.org
- IEEE Computer Society: http://www.computer.org
- Association for Computing Machinery (ACM): http://www.acm.org/
Homework 2, due 1 week later

- Re-do (optional, no extra credits) all examples in MIPS ISA (refer to CO 4, Chapter 2)
- Homework; see course Web site
- Reading assignment; see course Web site