Virtual Memory: Topics

- Why virtual memory?
- Virtual to physical address translation
- Page Table
- Translation Lookaside Buffer (TLB)
Another View of Memory Hierarchy

Upper Level
Faster

Upper Level
Larger
Lower Level

Thus far

Next: Virtual Memory

Upper Level
Faster

Upper Level
Larger
Lower Level

Why Virtual Memory?

- Today computers run multiple processes, each with its own address space
- Too expensive to dedicate a full-address-space worth of memory for each process
- Principle of Locality
  - allows caches to offer speed of cache memory with size of DRAM memory
  - DRAM can act as a “cache” for secondary storage (disk) ⇒ Virtual Memory
- Virtual memory – divides physical memory into blocks and allocate them to different processes
Virtual Memory Motivation

- Historically virtual memory was invented when programs became too large for physical memory.
- Allows OS to share memory and protect programs from each other (main reason today).
- Provides illusion of very large memory:
  - sum of the memory of many jobs greater than physical memory
  - allows each job to exceed the size of physical memory.
- Allows available physical memory to be very well utilized.
- Exploits memory hierarchy to keep average access time low.

Mapping Virtual to Physical Memory

- Program with 4 pages (A, B, C, D).
- Any chunk of Virtual Memory assigned to any chunk of Physical Memory (“page”).
Virtual Memory Terminology

- **Virtual Address**
  - address used by the programmer; CPU produces virtual addresses
- **Virtual Address Space**
  - collection of such addresses
- **Memory (Physical or Real) Address**
  - address of word in physical memory
- **Memory mapping or address translation**
  - process of virtual to physical address translation
- **More on terminology**
  - Page or Segment ↔ Block
  - Page Fault or Address Fault ↔ Miss

Comparing the 2 levels of hierarchy

<table>
<thead>
<tr>
<th>Parameter</th>
<th>L1 Cache</th>
<th>Virtual Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block/Page</td>
<td>16B – 128B</td>
<td>4KB – 64KB</td>
</tr>
<tr>
<td>Hit time</td>
<td>1 – 3 cc</td>
<td>50 – 150 cc</td>
</tr>
<tr>
<td>Miss Penalty (Access time)</td>
<td>8 – 150 cc</td>
<td>1M – 10M cc (Page Fault )</td>
</tr>
<tr>
<td>(Transfer time)</td>
<td>6 – 130 cc</td>
<td>800K – 8M cc</td>
</tr>
<tr>
<td></td>
<td>2 – 20 cc</td>
<td>200K – 2M cc</td>
</tr>
<tr>
<td>Hit Rate</td>
<td>0.1 – 10%</td>
<td>0.00001 – 0.001%</td>
</tr>
<tr>
<td>Placement:</td>
<td>DM or N-way SA</td>
<td>Fully associative (OS allows pages to be placed anywhere in main memory)</td>
</tr>
<tr>
<td>Address Mapping</td>
<td>25-45 bit physical address to 14-20 bit cache address</td>
<td>32-64 bit virtual address to 25-45 bit physical address</td>
</tr>
<tr>
<td>Replacement:</td>
<td>LRU or Random (HW cntr.)</td>
<td>LRU (SW controlled)</td>
</tr>
<tr>
<td>Write Policy</td>
<td>WB or WT</td>
<td>WB</td>
</tr>
</tbody>
</table>
Paging vs. Segmentation

- Two classes of virtual memory
  - Pages - fixed size blocks (4KB – 64KB)
  - Segments - variable size blocks (1B – 64KB/4GB)
  - Hybrid approach: Paged segments – a segment is an integral number of pages

<table>
<thead>
<tr>
<th></th>
<th>Paging</th>
<th>Segmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Paging vs. Segmentation: Pros and Cons

<table>
<thead>
<tr>
<th></th>
<th>Page</th>
<th>Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Words per address</td>
<td>One</td>
<td>Two (segment + offset)</td>
</tr>
<tr>
<td>Programmer visible?</td>
<td>Invisible to AP</td>
<td>May be visible to AP</td>
</tr>
<tr>
<td>Replacing a block</td>
<td>Trivial (all blocks are the same size)</td>
<td>Hard (must find contiguous, variable-size unused portion)</td>
</tr>
<tr>
<td>Memory use inefficiency</td>
<td>Internal fragmentation (unused portion of page)</td>
<td>External fragmentation (unused pieces of main memory)</td>
</tr>
<tr>
<td>Efficient disk traffic</td>
<td>Yes (adjust page size to balance access time and transfer time)</td>
<td>Not always (small segments transfer few bytes)</td>
</tr>
</tbody>
</table>

07/02/2005 UAH-CPE631
Each program operates in its own virtual address space
- Each is protected from the other
- OS can decide where each goes in memory
- Combination of HW + SW provides virtual → physical mapping

Virtual Memory Mapping Function
- Use table lookup ("Page Table") for mappings: Virtual Page number is index
- Virtual Memory Mapping Function
  - Physical Offset = Virtual Offset
  - Physical Page Number (P.P.N. or "Page frame")
    = PageTable[Virtual Page Number]
A page table is an operating system structure which contains the mapping of virtual addresses to physical locations

- There are several different ways, all up to the operating system, to keep this data around

- Each process running in the operating system has its own page table
  - “State” of process is PC, all registers, plus page table
  - OS changes page tables by changing contents of Page Table Base Register
Page Table Entry (PTE) Format

- Valid bit indicates if page is in memory
  - OS maps to disk if Not Valid (V = 0)
- Contains mappings for every possible virtual page

<table>
<thead>
<tr>
<th>V.</th>
<th>A.R.</th>
<th>P.P.T.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid</td>
<td>Access Rights</td>
<td>Physical Page Number</td>
</tr>
<tr>
<td>V.</td>
<td>A.R.</td>
<td>P.P.T</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>....</td>
</tr>
</tbody>
</table>

- If valid, also check if have permission to use page:
  Access Rights (A.R.) may be
  Read Only, Read/Write, Executable

Virtual Memory Problem #1

- Not enough physical memory!
  - Only, say, 64 MB of physical memory
  - N processes, each 4GB of virtual memory!
  - Could have 1K virtual pages/physical page!
- Spatial Locality to the rescue
  - Each page is 4 KB, lots of nearby references
  - No matter how big program is,
    at any time only accessing a few pages
  - "Working Set": recently used pages
VM Problem #2: Fast Address Translation

- PTs are stored in main memory
  ⇒ Every memory access logically takes at least twice as long, one access to obtain physical address and second access to get the data
- Observation: locality in pages of data, must be locality in virtual addresses of those pages
  ⇒ Remember the last translation(s)
- Address translations are kept in a special cache called Translation Look-Aside Buffer or TLB
- TLB must be on chip; its access time is comparable to cache

Typical TLB Format

<table>
<thead>
<tr>
<th>Virtual Addr.</th>
<th>Physical Addr.</th>
<th>Dirty</th>
<th>Ref</th>
<th>Valid</th>
<th>Access Rights</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Tag: Portion of virtual address
- Data: Physical Page number
- Dirty: since use write back, need to know whether or not to write page to disk when replaced
- Ref: Used to help calculate LRU on replacement
- Valid: Entry is valid
- Access rights: R (read permission), W (write perm.)
Translation Look-Aside Buffers

- TLBs usually small, typically 128 - 256 entries
- Like any other cache, the TLB can be fully associative, set associative, or direct mapped

TLB Translation Steps

- Assume 32 entries, fully-associative TLB (Alpha AXP 21064)
- 1: Processor sends the virtual address to all tags
- 2: If there is a hit (there is an entry in TLB with that Virtual Page number and valid bit is 1) and there is no access violation, then
- 3: Matching tag sends the corresponding Physical Page number
- 4: Combine Physical Page number and Page Offset to get full physical address
What if not in TLB?

- Option 1: Hardware checks page table and loads new Page Table Entry into TLB
- Option 2: Hardware traps to OS, up to OS to decide what to do
  - When in the operating system, we don't do translation (turn off virtual memory)
  - The operating system knows which program caused the TLB fault, page fault, and knows what the virtual address desired was requested
  - So it looks the data up in the page table
  - If the data is in memory, simply add the entry to the TLB, evicting an old entry from the TLB

What if the data is on disk?

- We load the page off the disk into a free block of memory, using a DMA transfer
  - Meantime we switch to some other process waiting to be run
- When the DMA is complete, we get an interrupt and update the process's page table
  - So when we switch back to the task, the desired data will be in memory
What if we don't have enough memory?

- We chose some other page belonging to a program and transfer it onto the disk if it is dirty
  - If clean (other copy is up-to-date), just overwrite that data in memory
  - We chose the page to evict based on replacement policy (e.g., LRU)
- And update that program's page table to reflect the fact that its memory moved somewhere else

Page Replacement Algorithms

- First-In/First Out
  - in response to page fault, replace the page that has been in memory for the longest period of time
  - does not make use of the principle of locality: an old but frequently used page could be replaced
  - easy to implement (OS maintains history thread through page table entries)
  - usually exhibits the worst behavior
- Least Recently Used
  - selects the least recently used page for replacement
  - requires knowledge of past references
  - more difficult to implement, good performance
Page Replacement Algorithms (cont’d)

- Not Recently Used
  (an estimation of LRU)
  - A reference bit flag is associated to each page table entry such that
    - Ref flag = 1 - if page has been referenced in recent past
    - Ref flag = 0 - otherwise
  - If replacement is necessary, choose any page frame such that its reference bit is 0
  - OS periodically clears the reference bits
  - Reference bit is set whenever a page is accessed

Selecting a Page Size

- Balance forces in favor of larger pages versus those in favoring smaller pages
  - Larger page
    - Reduce size PT (save space)
    - Larger caches with fast hits
    - More efficient transfer from the disk or possibly over the networks
    - Less TLB entries or less TLB misses
  - Smaller page
    - better conserve space, less wasted storage (Internal Fragmentation)
    - shorten startup time, especially with plenty of small processes
VM Problem #3: Page Table too big!

- Example
  - 4GB Virtual Memory ÷ 4 KB page
    => ~ 1 million Page Table Entries
    => 4 MB just for Page Table for 1 process,
    25 processes => 100 MB for Page Tables!
- Problem gets worse on modern 64-bits machines
- Solution is Hierarchical Page Table

Page Table Shrink

- Single Page Table Virtual Address
  - Page Number | Offset
    - 20 bits | 12 bits

- Multilevel Page Table Virtual Address
  - Super Page Number | Page Number | Offset
    - 10 bits | 10 bits | 12 bits

- Only have second level page table for valid entries of super level page table
  - If only 10% of entries of Super Page Table are valid, then total mapping size is roughly 1/10-th of single level page table
2-level Page Table

Physical Memory

Virtual Memory

2nd Level Page Tables

Super PageTable

Stack

Heap

Static

Code

The Big Picture

Virtual address

TLB access

No

TLB hit?

Yes

try to read from PT

No

try to read from cache

Write?

Yes

Set in TLB

No

Cache hit?

Yes

cache/buffer mem. write

No

cache miss stall

Deliver data to CPU

replace page from disk

page fault?

Yes

try to read from cache
The Big Picture (cont’d)
L1-8K, L2-4M, Page-8K, cl-64B, VA-64b, PA-41b

Things to Remember

- Apply Principle of Locality Recursively
- Manage memory to disk? Treat as cache
  - Included protection as bonus, now critical
  - Use Page Table of mappings vs. tag/data in cache
- Spatial locality means Working Set of pages is all that must be in memory for process to run
- Virtual memory to Physical Memory Translation too slow?
  - Add a cache of Virtual to Physical Address Translations, called a TLB
- Need more compact representation to reduce memory size cost of simple 1-level page table (especially 32 ⇒ 64-bit address)
Instruction Set Principles and Examples

Outline

- What is Instruction Set Architecture?
- Classifying ISA
- Elements of ISA
  - Programming Registers
  - Type and Size of Operands
  - Addressing Modes
  - Types of Operations
  - Instruction Encoding
- Role of Compilers
Shift in Applications Area

- **Desktop Computing** – emphasizes performance of programs with integer and floating point data types; little regard for program size or processor power
- **Servers** - used primarily for database, file server, and web applications; FP performance is much less important for performance than integers and strings
- **Embedded applications** value cost and power, so code size is important because less memory is both cheaper and lower power
- **DSPs and media processors**, which can be used in embedded applications, emphasize real-time performance and often deal with infinite, continuous streams of data
  - Architects of these machines traditionally identify a small number of key kernels that are critical to success, and hence are often supplied by the manufacturer.

What is ISA?

- **Instruction Set Architecture** – the computer visible to the assembler language programmer or compiler writer
- ISA includes
  - Programming Registers
  - Operand Access
  - Type and Size of Operands
  - Instruction Set
  - Addressing Modes
  - Instruction Encoding
Classifying ISA

- Stack Architectures -
  operands are implicitly on the top of the stack

- Accumulator Architectures -
  one operand is implicitly accumulator

- General-Purpose Register Architectures -
  only explicit operands,
  either registers or memory locations
  - register-memory:
    access memory as part of any instruction
  - register-register:
    access memory only with load and store instructions

Classifying ISA (cont’d)

For classes: Stack, Accumulator, Register-Memory, Load-store (or Register-Register)
Example: Code Sequence for $C = A + B$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>Register-Memory</th>
<th>Load-store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Load A</td>
<td>Load R1, A</td>
<td>Load R1, A</td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td>Add R3, R1, B</td>
<td>Load R2, B</td>
</tr>
<tr>
<td>Add</td>
<td></td>
<td>Add R3, R1, R2</td>
<td></td>
</tr>
<tr>
<td>Pop C</td>
<td>Store C</td>
<td>Store C, R3</td>
<td>Store C, R3</td>
</tr>
</tbody>
</table>

4 instr. 3 mem. op. 3 instr. 3 mem. op. 3 instr. 3 mem. op. 4 instr. 3 mem. op.

Development of ISA

- Early computers used stack or accumulator architectures
  - accumulator architecture easy to build
  - stack architecture closely matches expression evaluation algorithms (without optimisations!)
- GPR architectures dominate from 1975
  - registers are faster than memory
  - registers are easier for a compiler to use
  - hold variables
    - memory traffic is reduced, and the program speedups
    - code density is increased
      (registers are named with fewer bits than memory locations)
Programming Registers

- Ideally, use of GPRs should be orthogonal; i.e., any register can be used as any operand with any instruction
- May be difficult to implement; some CPUs compromise by limiting use of some registers
- How many registers?
  - PDP-11: 8; some reserved (e.g., PC, SP); only a few left, typically used for expression evaluation
  - VAX 11/780: 16; some reserved (e.g., PC, SP, FP); enough left to keep some variables in registers
  - RISC: 32; can keep many variables in registers

Operand Access

- Number of operands
  - 3; instruction specifies result and 2 source operands
  - 2; one of the operands is both a source and a result
- How many of the operands may be memory addresses in ALU instructions?

<table>
<thead>
<tr>
<th>Number of memory addresses</th>
<th>Maximum number of operands</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>SPARC, MIPS, HP-PA, PowerPC, Alpha, ARM, Trimedia</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Intel 80x86, Motorola 68000, TI TMS320C54</td>
</tr>
<tr>
<td>2/3</td>
<td>2/3</td>
<td>VAX</td>
</tr>
</tbody>
</table>
**Operand Access: Comparison**

<table>
<thead>
<tr>
<th>Type</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reg-Reg (0-3)</td>
<td>Simple, fixed length instruction encoding. Simple code instruction.</td>
<td>Higher inst. count. Some instructions are short and bit encoding may be wasteful.</td>
</tr>
<tr>
<td>Reg-Mem (1,2)</td>
<td>Data can be accessed without loading first. Instruction format tends to be easy to decode and yields good density.</td>
<td>Source operand is destroyed in a binary operation. Clocks per instruction varies by operand location.</td>
</tr>
<tr>
<td>Mem-Mem (3,3)</td>
<td>Most compact.</td>
<td>Large variation in instruction size and clocks per instructions. Memory bottleneck.</td>
</tr>
</tbody>
</table>

**Type and Size of Operands**

- How is the type of an operand designated?
  - encoded in the opcode; most often used (eg. Add, AddU)
  - data are annotated with tags that are interpreted by hw

- Common operand types
  - character (1 byte) {ASCII}
  - half word (16 bits)
    {short integers, 16-bit Java Unicode}
  - word (32 bits) {integers}
  - single-precision floating point (32 bits)
  - double-precision floating point (64 bits)
  - binary packed/unpacked decimal - used infrequently
Type and Size of Operands (cont’d)

- Distribution of data accesses by size (SPEC)
  - Double word: 0% (Int), 69% (Fp)
  - Word: 74% (Int), 31% (Fp)
  - Half word: 19% (Int), 0% (Fp)
  - Byte: 7% (Int), 0% (Fp)

- Summary:
  a new 32-bit architecture should support
  - 8-, 16- and 32-bit integers; 64-bit floats
  - 64-bit integers may be needed for 64-bit addressing
  - others can be implemented in software

- Operands for media and signal processing
  - Pixel – 8b (red), 8b (green), 8b (blue), 8b (transparency of the pixel)
  - Fixed-point (DSP) – cheap floating-point
  - Vertex (graphic operations) – x, y, z, w

Addressing Modes

- Addressing mode - how a computer system specifies the address of an operand
  - constants
  - registers
  - memory locations
  - I/O addresses

- Memory addressing
  - since 1980 almost every machine uses addresses to level of 1 byte =>
  - How do byte addresses map onto 32 bits word?
  - Can a word be placed on any byte boundary?
Interpreting Memory Addresses

- **Big Endian**
  - address of most significant byte = word address (xx00 = Big End of the word);
  - IBM 360/370, MIPS, Sparc, HP-PA

- **Little Endian**
  - address of least significant byte = word address (xx00 = Little End of the word);
  - Intel 80x86, DEC VAX, DEC Alpha

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

### Big Endian

<table>
<thead>
<tr>
<th>Memory</th>
<th>Big Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0x00</td>
</tr>
<tr>
<td>a+1</td>
<td>0x01</td>
</tr>
<tr>
<td>a+2</td>
<td>0x02</td>
</tr>
<tr>
<td>a+3</td>
<td>0x03</td>
</tr>
</tbody>
</table>

### Little Endian

<table>
<thead>
<tr>
<th>Memory</th>
<th>Little Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0x00</td>
</tr>
<tr>
<td>a+1</td>
<td>0x01</td>
</tr>
<tr>
<td>a+2</td>
<td>0x02</td>
</tr>
<tr>
<td>a+3</td>
<td>0x03</td>
</tr>
</tbody>
</table>

**Aligned**

**Not Aligned**
### Addressing Modes: Examples

<table>
<thead>
<tr>
<th>Addr. mode</th>
<th>Example</th>
<th>Meaning</th>
<th>When used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>ADD R4, R3</td>
<td>Regs[R4] ← Regs[R4] + Regs[R3]</td>
<td>a value is in register</td>
</tr>
<tr>
<td>Immediate</td>
<td>ADD R4, #3</td>
<td>Regs[R4] ← Regs[R4] + 3</td>
<td>for constants</td>
</tr>
</tbody>
</table>

### Addressing Mode Usage

- **3 programs measured on machine with all address modes (VAX)**
  - register direct modes are not counted (one-half of the operand references)
  - PC-relative is not counted (exclusively used for branches)
- **Results**
  - Displacement: 42% avg, (32 - 55%)
  - Immediate: 33% avg, (17 - 43%)
  - Register indirect: 13% avg, (3 - 24%)
  - Scaled: 7% avg, (0 - 16%)
  - Memory indirect: 3% avg, (1 - 6%)
  - Misc.: 2% avg, (0 - 3%)

![Graph](image)
Displacement, immediate size

- Displacement
  - 1% of addresses require > 16 bits
  - 25% of addresses require > 12 bits

- Immediate
  - If they need to be supported by all operations?
    - Loads: 10% (Int), 45% (Fp)
    - Compares: 87% (Int), 77% (Fp)
    - ALU operations: 58% (Int), 78% (Fp)
    - All instructions: 35% (Int), 10% (Fp)
  - What is the range of values?
    - 50% - 70% fit within 8 bits
    - 75% - 80% fit within 16 bits

Addressing modes: Summary

- Data addressing modes that are important:
  Displacement, Immediate, Register Indirect
- Displacement size should be 12 to 16 bits
- Immediate should be 8 to 16 bits
### Addressing Modes for Signal Processing

- DSPs deal with continuous infinite stream of data => circular buffers
  - Modulo or Circular addressing mode
- FFT shuffles data at the start or end
  - 0 (000) => 0 (000), 1 (001) => 4 (100), 2 (010) => 2 (010), 3 (011) => 6 (110), ...
  - Bit reverse addressing mode
    - take original value, do bit reverse, and use it as an address
- 6 mfu modes from found in desktop, account for 95% of the DSP addr. modes

### Typical Operations

- **Data Movement**: load (from memory), store (to memory), mem-to-mem move, reg-to-reg move, input (from IO device), push (to stack), pop (from stack), output (to IO device),
- **Arithmetic**: integer (binary + decimal), Add, Subtract, Multiply, Divide
- **Shift**: shift left/right, rotate left/right
- **Logical**: not, and, or, xor, clear, set
- **Control**: unconditional/conditional jump, call/return
- **Subroutine Linkage**: OS call, virtual memory management
- **System**: test-and-set
- **Floating-point**: FP Add, Subtract, Multiply, Divide, Compare, SQRT
- **String**: String move, compare, search
- **Graphics**: Pixel and vertex operations, compression/decomp.
Top ten 8086 instructions

<table>
<thead>
<tr>
<th>Rank</th>
<th>Instruction</th>
<th>% total execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load</td>
<td>22%</td>
</tr>
<tr>
<td>2</td>
<td>conditional branch</td>
<td>20%</td>
</tr>
<tr>
<td>3</td>
<td>compare</td>
<td>16%</td>
</tr>
<tr>
<td>4</td>
<td>store</td>
<td>12%</td>
</tr>
<tr>
<td>5</td>
<td>add</td>
<td>8%</td>
</tr>
<tr>
<td>6</td>
<td>and</td>
<td>6%</td>
</tr>
<tr>
<td>7</td>
<td>sub</td>
<td>5%</td>
</tr>
<tr>
<td>8</td>
<td>move reg-reg</td>
<td>4%</td>
</tr>
<tr>
<td>9</td>
<td>call</td>
<td>1%</td>
</tr>
<tr>
<td>10</td>
<td>return</td>
<td>1%</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>96%</td>
</tr>
</tbody>
</table>

Simple instructions dominate instruction frequency
=> support them

Operations for Media and Signal Processing

- Multimedia processing and limits of human perception
  - use narrower data words (don’t need 64b fp)
    => wide ALU’s operate on several data items at the same time
  - partition add – e.g., perform four 16-bit adds on a 64-bit ALU
  - SIMD – Single instruction Multiple Data or vector instructions (see Appendix F)
    - Figure 2.17 (page 110)

- DSP processors
  - algorithms often need saturating arithmetic
    - if result too large to be represented, it is set to the largest representable number
  - often need several rounding modes
    - MAC (Multiply and Accumulate)
Instructions for Control Flow

- Control flow instructions
  - Conditional branches (75% int, 82% fp)
  - Call/return (19% int, 8% fp)
  - Jump (6% int, 10% fp)

- Addressing modes for control flows
  - PC-relative
  - for returns and indirect jumps the target is not known in compile time => specify a register which contains the target address

Instructions for Control Flow (cont’d)

- Methods for branch evaluation
  - Condition Code – CC (ARM, 80x86, PowerPC)
    - tests special bits set by ALU instructions
  - Condition register (Alpha, MIPS)
    - tests arbitrary register
  - Compare and branch (PA-RISC, VAX)
    - compare is a part of branch

- Procedure invocation options
  - do control transfer and possibly some state saving
    - at least return address must be saved (in link register)
  - compiler generate loads and stores to save the state
  - Caller savings vs. callee savings
Encoding an Instruction Set

- Instruction set architect must choose how to represent instructions in machine code
  - Operation is specified in one field called Opcode
  - Each operand is specified by a separate Address specifier (tells which addressing modes is used)

- Balance among
  - Many registers and addressing modes adds to richness
  - Many registers and addressing modes increase code size
  - Lengths of code objects should "match" architecture; e.g., 16 or 32 bits

Basic variations in encoding

a) Variable (e.g. VAX)

- Operation & no. of operands
- Address specifier 1
- Address field 1
- ... Address specifier n
- Address field n

b) Fixed (e.g. DLX, MIPS, PowerPC, ...)

- Operation
- Address field 1
- Address field 2
- Address field 3

c) Hybrid (e.g. IBM 360/70, Intel80x86)

- Operation
- Address specifier 1
- Address field 1

- Operation
- Address specifier 1
- Address specifier 2
- Address field

- Operation
- Address specifier
- Address field 1
- Address field 2
Summary of Instruction Formats

- If code size is most important, use variable length instructions
- If performance is over most important, use fixed length instructions
- Reduced code size in RISCs
  - hybrid version with both 16-bit and 32-bit ins.
    - narrow instructions support fewer operations, smaller address and immediate fields, fewer registers, and 2-address format
    - ARM Thumb, MIPS MIPS16 (Appendix C)
  - IBM: compressed code is kept in main memory, ROMs, disk
    - caches keep decompressed code

Role of Compilers

- Structure of recent compilers
  - 1) Front-end
    - transform language to common intermediate form
      - language dependent, machine independent
  - 2) High-level optimizations
    - e.g., loop transformations, procedure inlining,...
      - somewhat language dependent, machine independent
  - 3) Global optimizer
    - global and local optimizations, register allocation
      - small language dependencies, somewhat machine dependencies
  - 4) Code generator
    - instruction selection, machine dependent optimizations
      - language independent, highly machine dependent
Compiler Optimizations

1) High-level optimizations
   – done on the source

2) Local optimizations
   – optimize code within a basic block

3) Global optimizations
   – extend local optimizations across branches (loops)

4) Register allocation
   – associates registers with operands

5) Processor-dependent optimizations
   – take advantage of specific architectural knowledge