Common addressing modes[H&P §2.3] How does an architecture specify an address? No single way is good for all situations, so all architectures support a variety of addressing modes.

Register
- ADD R4, R3
- R4 = R4 + R3
- Used when value is in a register

Immediate
- ADD R4, #3
- R4 = R4 + 3
- Useful for small constants, which occur frequently

Displacement
- ADD R4, 100(R1)
- R4 = R4 + MEM[100+R1]
- Can be used to access a stack frame (which contains arguments and local variables).
- Can be used to access the global data segment (see pp. 124–125 of H&P).
- Can be used to access fields of a record (a C struct or a C++ class object).

Register deferred/Register indirect
- ADD R3, (R1)
- R3 = R3 + MEM[R1]
- Access using a pointer or a computed address

Indexed
- Add R3, (R1 + R2)
- R3 = R3 + Mem[R1 + R2]
- Used for array accesses (R1 = base, R2 = index).

Direct/Absolute
- ADD R1, (1001)
• $R1 = R1 + M[1001]$  
  Used for accessing global ("static") data

**Memory indirect/Memory deferred**

• ADD $R1, @(R3)$  
  $R1 = R1 + MEM[MEM[R3]]$  
  Used for pointer dereferencing: $x = *p$; (if $p$ is not register-allocated)

**Autoincrement/Postincrement**

• ADD $R1, (R2)+$  
  $R1 = R1 + MEM[R2]; R2 = R2 + d$ ($d$ is size of operand)  
  Used for looping through arrays, stack pop

**Autodecrement/Predecrement**

• ADD $R1, -(R2)$  
  $R2 = R2 - d; R1 = R1 + MEM[R2]$ ($d$ is size of operand)  
  Same uses as autoincrement, stack push

**Scaled**

• ADD $R1, 100(R2)[R3]$  
  $R1 = R1 + MEM[100+R2+R3*d]$ ($d$ is size of operation)  
  Used for accessing arrays with non-byte-sized elements.

Which modes are used most frequently? H&P show a table based on three benchmarks run on a VAX architecture:
Three SPEC 89 programs were measured for this graph.

Two addressing modes dominate, and five modes are responsible for all but 0% to 3% of references.

This graph shows memory addresses, not register numbers (about half of all operands are in registers).

Most displacements are small, but enough of them are large to raise the average for floating-point programs to 13 bits:
Why should we care about what values are used for displacements?

Similarly, if we measure the length of immediates, we get this distribution:
These measurements were taken on an Alpha, where the maximum immediate is 16 bits. On the VAX, which supports 32-bit immediates, 20%–25% of immediates are longer than 16 bits.

**Wisdom about modes**

Need:

- Register, Displacement, Immediate and optionally Indexed (indexed simplifies array accesses)
- Displacement size 12–16 bits (empirical)
- Immediate: 8 to 16 bits (empirical)
- Can synthesize the rest from simpler instructions
- Example—MIPS architecture:
  - Register, displacement, Immediate modes only
  - both immediate and displacement: 16 bits

Choice depends on workload!

- For example, floating-point codes might require larger immediates, or 64-bit word-size machines might also require larger immediates (for *p++ kind of operations)

**Exercise:** Think for a moment about the tradeoff between larger and smaller fields for displacements and immediates. What are the advantages and disadvantages of using longer fields for them?

**Control transfer semantics**

- Conditional
- Unconditional
  - Normal
  - Call
  - Return

Why are Call and Return instructions different from ordinary jumps?
This graph shows the relative frequencies of these kinds of instructions.

![Graph showing instructions frequencies](image)

**PC-Relative vs. Absolute**

- Branch allows relocatable ("position independent") code.
- Jump allows branching further than PC-relative

How long should our branch displacement be? Let’s take a look at the number of instructions separating branch instructions from their targets.

![Graph showing branch displacement](image)

Eight bits, therefore, is enough to encode most branch targets.
Register-indirect jumps

Neither of the above two addressing modes (PC-relative or absolute) will work for Returns, or other situations in which the target is not known at compile time.

Two choices:

• Make the jump go to a location specified by the contents of a register.
• Allow any addressing mode to be used to specify the jump target.

When do we have jumps whose target is not known at compile time?

• Procedure returns.
• case (or switch) statements.
• Method (or virtual function) calls.
• Function pointers (pass a function as a parameter).
• Dynamic linking of libraries. Such libraries are loaded and linked only when they are invoked by the program (good for error routines, etc.).

Parts of a control transfer
WHERE: Determine the target address.
WHETHER: Determine if transfer should occur.
WHEN: Determine when in time the transfer should occur.

Each of the three decisions can be decoupled from the others.

All three together: Compare and branch instruction

• BR (R1 = R2), destination
• (+) A single instruction
• (−) Heavy hardware requirement, inflexible scheduling
WHETHER separate from WHERE/WHEN:

Condition-code register (CMP R1,R2; …; BEQ dest)(+)
Sometimes test happens “for free,” as when it is set by
another ALU operation (e.g., +, −, *, /)

(−) Hard for compiler to figure out which instructions depend
on CC register

Condition register (SUB R1,R2; …; BEQ R1, dest)

(+ ) Simple to implement, dependencies between instructions
are obvious to compiler

(−) Uses a register (“register pressure”)

Prepare-to-branchDecouple all three of WHERE/WHETHER/WHEN

WHERE: PBR BTR1 = destination

(BTR1 = “Branch target register #1”)

WHETHER: CMP PR2 = (R1 = R2)

(PR2 = “Predicate register #2”)

WHEN: BR BTR1 if PR2

(+ ) Schedule each instruction so it happens during “free time”
when hardware is idle.

(−) Three instructions: higher IC.

From the HP Labs PlayDoh architecture

Instruction Encoding tradeoffs[H&P §2.10] Variable width

<table>
<thead>
<tr>
<th>Operation &amp; # of operands</th>
<th>Address specifier 1</th>
<th>Address field 1</th>
<th>…</th>
<th>Address specifier n</th>
<th>Address field n</th>
</tr>
</thead>
</table>

• Common instructions are short (1-2 bytes), less common or
more complex instructions are long (>2 bytes)

(+ ) Very versatile, uses memory efficiently
(-) Instruction words must be decoded before number of instructions is known

Fixed width

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address field 1</th>
<th>Address field 2</th>
<th>Address field 3</th>
</tr>
</thead>
</table>

- Typically 1 instruction per 32-bit word (Alpha is 2 instructions per word)

(+) Every instruction word is an instruction, Easier to fetch/decode

(-) Uses memory inefficiently

A hybrid instruction set is also possible

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address specifier</th>
<th>Address field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Address specifier 1</td>
<td>Address specifier 2</td>
</tr>
<tr>
<td>Operation</td>
<td>Address specifier</td>
<td>Address field 1</td>
</tr>
</tbody>
</table>

**Addressing mode encoding** Each operand has a “mode” field

- Also called “address specifiers”
- VAX, 68000

(+) Very versatile

(-) Encourages variable-width instructions (hard decode)

Opcode specifies addressing mode

- Most RISCs

(+) Encourages fixed-width instructions (easy decode).

(+) “Natural” for a load/store ISA.
(-) Limits what every instruction can do (but this only matters for loads and stores).

**Compiler impact** [H&P §2.11]

**High-level opt:**
- Use a “virtual source level” representation
- Loop interchange, etc.
- Largely machine independent

**Low-level opt:**
- Each “optimization pass” runs as a filter
- Local and global optimization, and register allocation.
- Enhances parallelism.
- Small machine dependencies

**Code generation:**
- Schedule code for high performance.
- More later on this.