Instruction Set Architecture

Chapter 3

Instruction Set Architecture

- ★ Computer Architecture class in the past.
- ★ Instruction Set Architecture
  - History, Matrix, Design Issues
- ★ Memory Organization
  - Endian
  - Alignment
- ★ Addressing Mode
- ★ Instruction Format
- ★ Roles of Compiler in Optimization
- ★ Parameter Passing
- ★ Exception

Computer Architecture Class in the past

- ★ 50s and 60s - Computer Arithmetic
- ★ 70s and 80s - Instruction Set Design and ISA for Compiler
  - Key evidences: Pascal, Ada, Stack Frame, etc
- ★ 90s - CPU Organization, Memory Architecture, Instruction Set Extension
  - Key evidences: Intel Architecture, Multimedia Extension, Nvidia (1993), ATI (1985)
- ★ 2000s - Computer Arithmetic, I/O system, Parallelism
  - Key evidences: multi-core processor, 64-bit architecture, PCI-express, USB, Thunderbolt
- ★ 2010s - GPU? (deep learning), Virtualization?, Distributed Architecture?
  - See trends around you.

Instruction Set Architecture

- ★ What do you need to know in order to program a computer machine in assembly programming language (i.e. to build system software such as O.S. , Compiler, Assembler, etc)?
  - Instructions or Operators - (Logical) Register Transfer Language, CPI, format
  - Operands, Registers and flags (status) - How the operators interact with data
  - Memory Reference - Addressing, Referencing and Dereferencing memory
  - Instruction Format - How an instruction is translated into binary code.
  - Interrupt and Exception Handler
- ★ This is basically called “Instruction Set Architecture”.
  (Reminder. ISA is abstraction between hardware and software.)
- ★ Note that we currently ignore external components.

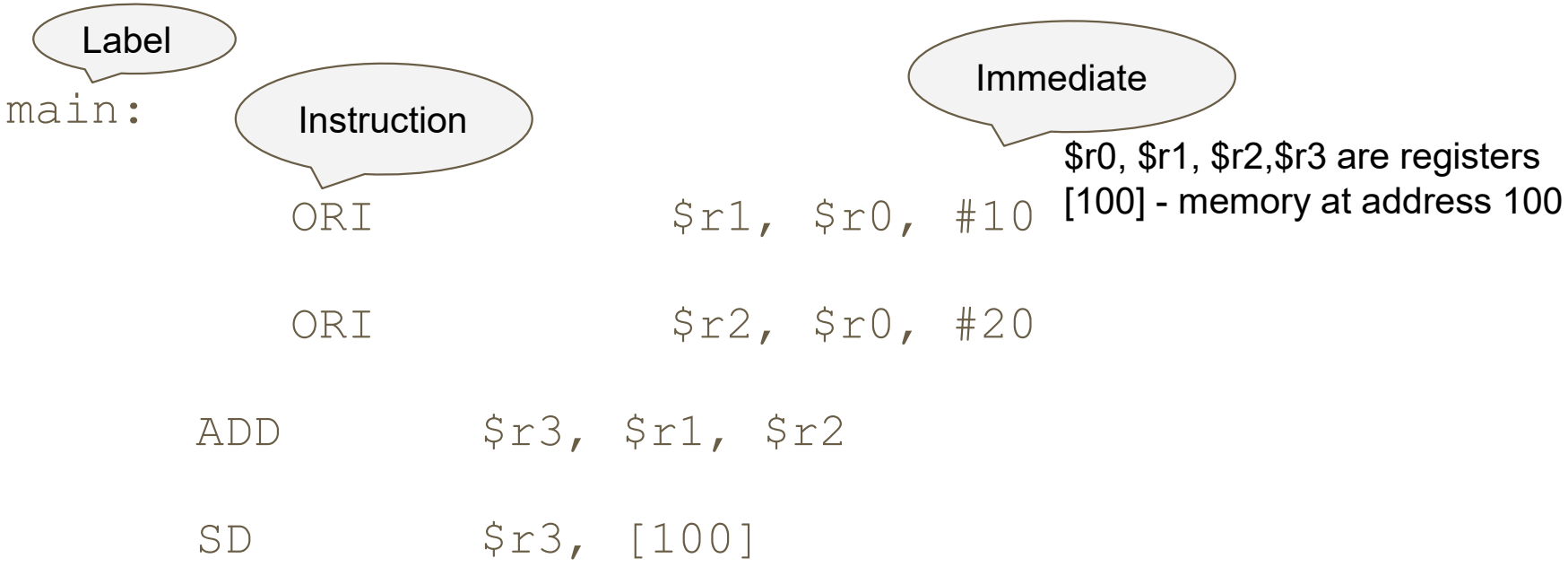
Instruction Set Architecture (ctd.)

- ★ In 1964, IBM introduces the IBM 360 architecture, which eliminated 7 different IBM instruction sets.
  - Before that, programs are not portable.

- ★ The instruction set architecture serves as the interface between software and hardware.
- ★ It provides the mechanism by which the software tells the hardware what should be done.

Anatomy of Assembly Programming Language

- ★ We will quickly visit assembly programming language.
- ★ C=10+20;



Design Issues

- ★ Where are operands stored?
  - Accumulator, stack, registers, memory
- ★ How many explicit operands?
  - 0, 1, 2, or 3
- ★ How is the operand be specified?
  - addressing mode (immediate, direct, indirect, displacement)

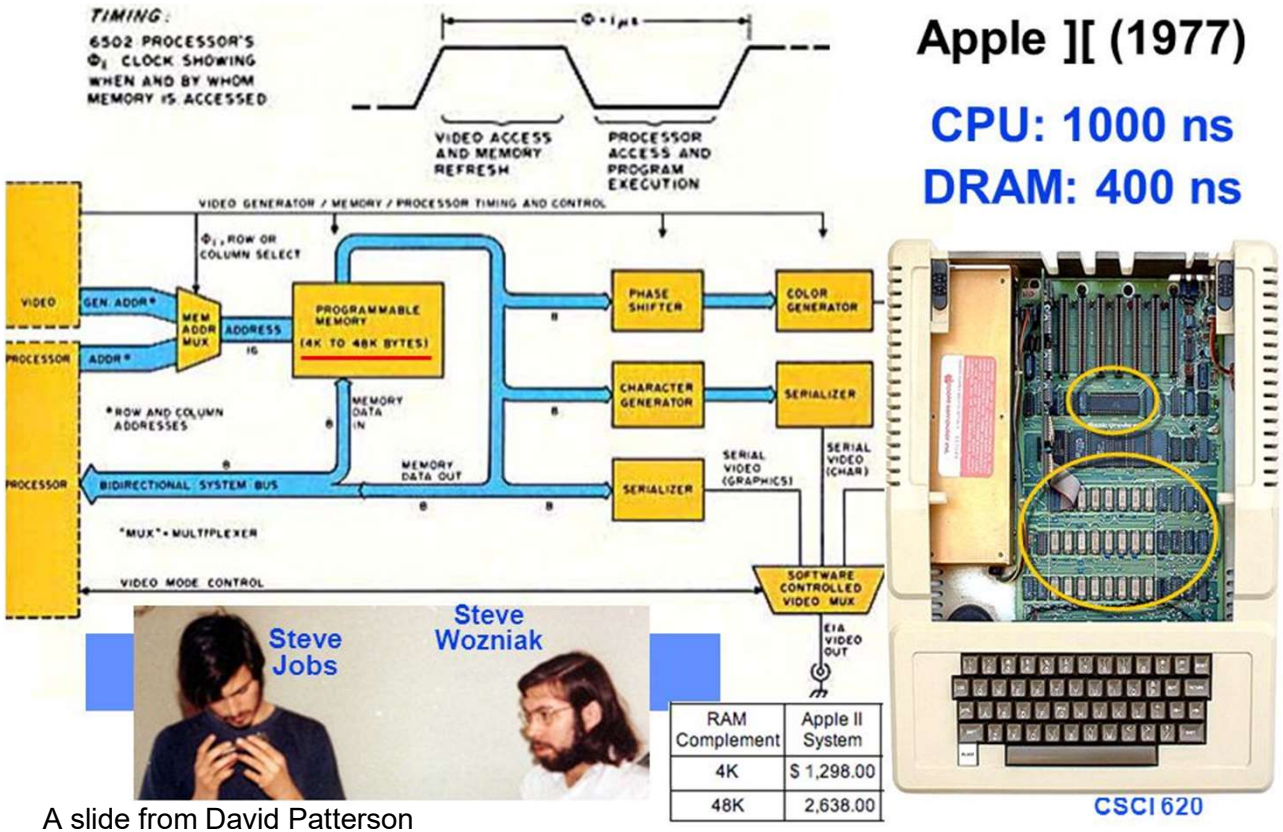
ISA Matrix

- ★ Properties of preferred ISA
  - Compatibility
    - Portable software
  - Generality / Completeness
    - Support a wide rage of operations
  - Orthogonal
    - Few special cases, all operands modes available with any data/instruction type
  - Regularity
    - No overloading for the meanings of instruction fields
  - Ease of Compilation
  - East of Implementation

History of ISA

- ★ Historically, memory (DRAM) was fast comparing to CPU.
- ★ Transistor is much more expensive.
- ★ Keep in mind that back then, technology are limited.

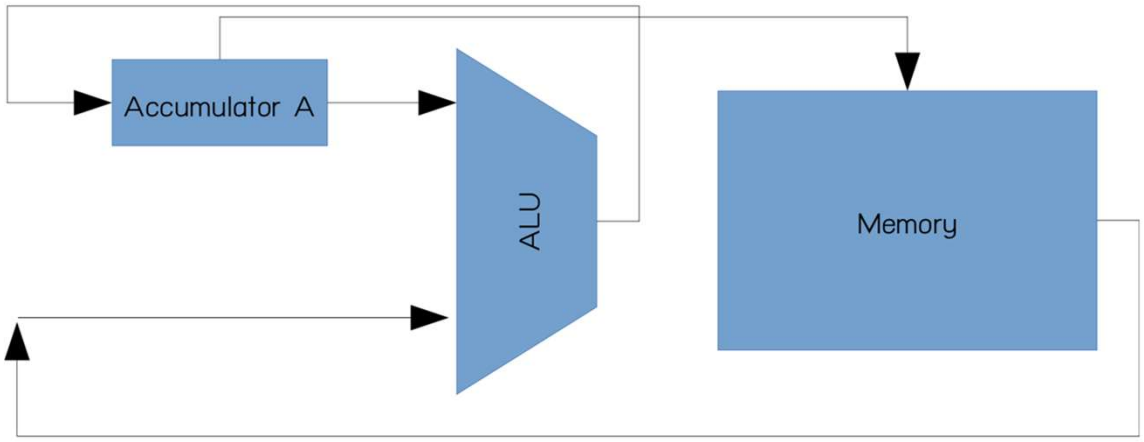
1977: At one time, DRAM was faster than microprocessors



A slide from David Patterson

Accumulator Architecture

- ★ Before 1960
- ★ 1 address. Accumulator is an implicit source and/or target
- ★ Only one register (accumulator) is enough for every calculations.
- ★ Remember your grade-school adder.



x=(a*b)+(a–(b*c))

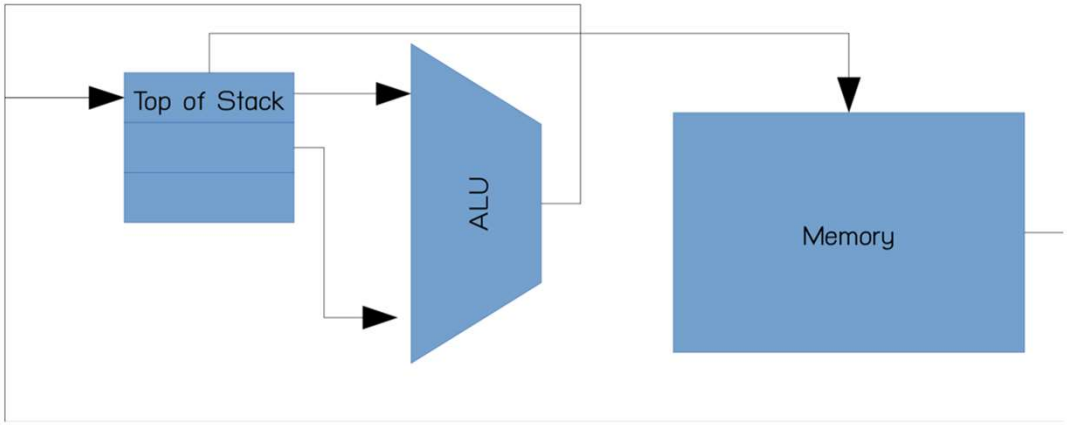
```
LD      $A, [a]   ;     $A=a
MUL     $A, [b]   ;     $A=a*b
SD      $A, [x]   ;     x=$A
LD      $A, [b]   ;     $A=b
MUL     $A, [c]   ;     $A=b*c
LD      $A, [a]   ;     $A=a
SUB     $A, [y]   ;     $A=a-(b*c)
SD      $A, [y]   ;     y=$A
LD      $A, [x]   ;     $A=x
ADD     $A, [y]   ;     $A=(a*b)+(a-
SD      $A, [x]   ;     x=$a
```

Accumulator Architecture (ctd.)

- 11 instructions
- Pros
  - (Very) low hardware requirements
  - Easy to design and understand
- Cons
  - Bottleneck (Accumulator)
  - Bad for parallelism (including pipelining)
  - High memory traffic

Stack Architecture

- ★ 1960s to 1970s
- ★ 0 address. Operands are the top of stack.



x=(a*b)+(a–(b*c))

```
PUSH    a     ; (top of stack) a
PUSH    b     ; (top of stack)  b , a
MUL           ; (top of stack) (a*b)
PUSH    a     ; (top of stack) a, (a*b)
PUSH    b     ; (top of stack) b, a, (a*b)
PUSH    c     ; (top of stack) c, b, a, (a*b)
MUL           ; (top of stack) (b*c), a, (a*b)
SUB           ; (top of stack) (a - (b*c) , (a*b)
ADD           ; (top of stack) (a*b) + (a - (b*c))
POP     x     ;
```

Stack Architecture (ctd.)

- ★ 10 instructions
- ★ Pros
  - Low hardware requirements
  - Good code density
  - Easy for compiler
- ★ Cons
  - Bottleneck (stack)
  - Required data is not always at the top of stack
  - Difficult for optimization (compiler)
  - Bad for parallelism (including pipelining)

Memory-Memory Architecture

x=(a*b)+(a–(b*c))

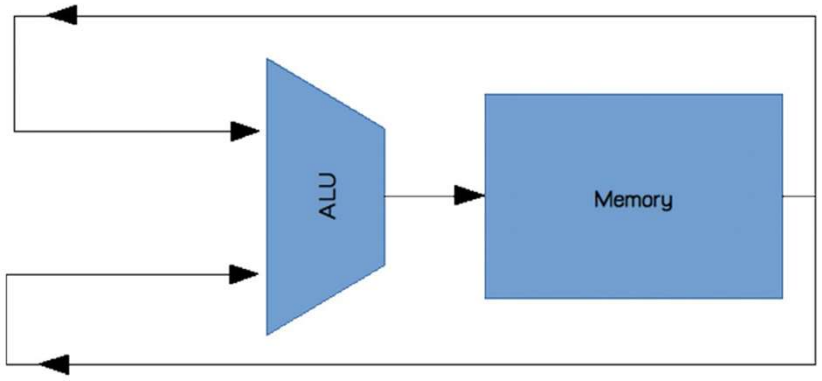
- ★ 1970s to 1980s
- ★ 2-3 address. Do every calculations on memory

3 operands

```
MUL       k, a, b   ; k = a*b
MUL       i, b, c   ; I = b*c
SUB       j, a, i   ; j = a-i = a - (b*c)
ADD       x, k, j   ; x = k+j = (a*b) + (a - (b*c))
```

2 operands

```
LD    x,a    ; x = a
MUL    x, b    ; x = x*b = a*b
LD    i, b    ; i = b
MUL    i, c    ; i = i*c = b*c
LD    j, a    ; j = a
SUB    j, i    ; j = j - i = a - (b*c)
ADD    x, j    ; x = x+j = (a*b) + (a - (b*c))
```

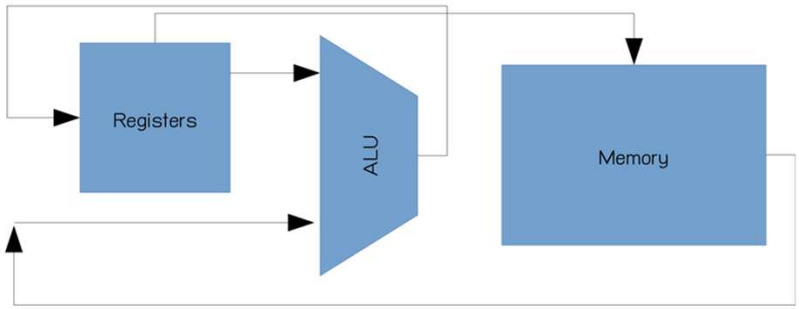


Memory-Memory Architecture (ctd.)

- ★ 4 instructions (3-operand version), 7 instructions (2-operand version)
- ★ Pros
  - Fewer instructions
  - Easy for compilers
- ★ Cons
  - (Very) high memory traffic
  - Variable number of CPI (due to memory access)
  - For 2-operand version, more data movements are required.

Register-Memory Architecture

- ★ 1970s to present
- ★ 2 address
- ★ Like accumulator, but more registers.



x=(a*b)+(a–(b*c))

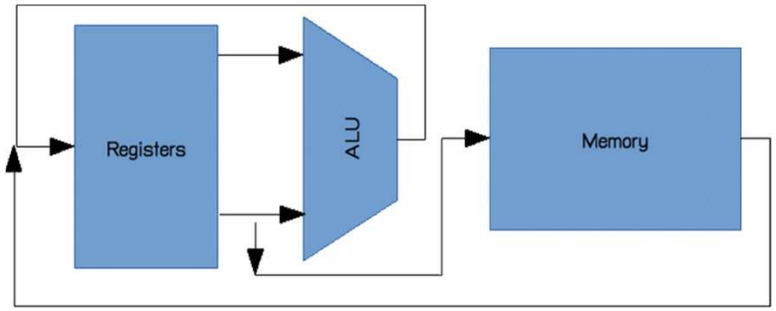
```
LD          $r1, a     ; $r1 = a
MUL         $r1, b     ; $r1 = $r1*b = a*b
LD          $r2, b     ; $r2 = b
MUL         $r2, c     ; $r2 = $r2*c = b*c
SD          $r2, j     ; j = $r2
LD     $r3, a     ; $r3 = a
SUB         $r3, j     ; $r3 = $r3-j = a - (b*c)
SD          $r3, j     ; j = $r3
ADD         $r1, j     ; $r1 = $r1-j
SD          $r1, x
```

Register-Memory Architecture (ctd.)

- ★ 10 instructions
- ★ Pros
  - ○ (Some) Data can be accessed without loading
  - ○ Easy to encode (instruction format)
  - ○ Good code density
- ★ Cons
  - ○ Poor orthogonal (operands are not equivalent)
  - ○ Variable number of CPI
  - ○ (May) limit number of registers

Register-Register Architecture

- ★ 1960s to present
- ★ 3 address
- ★ Operands are registers only.
- ★ Aka. Load-Store architecture



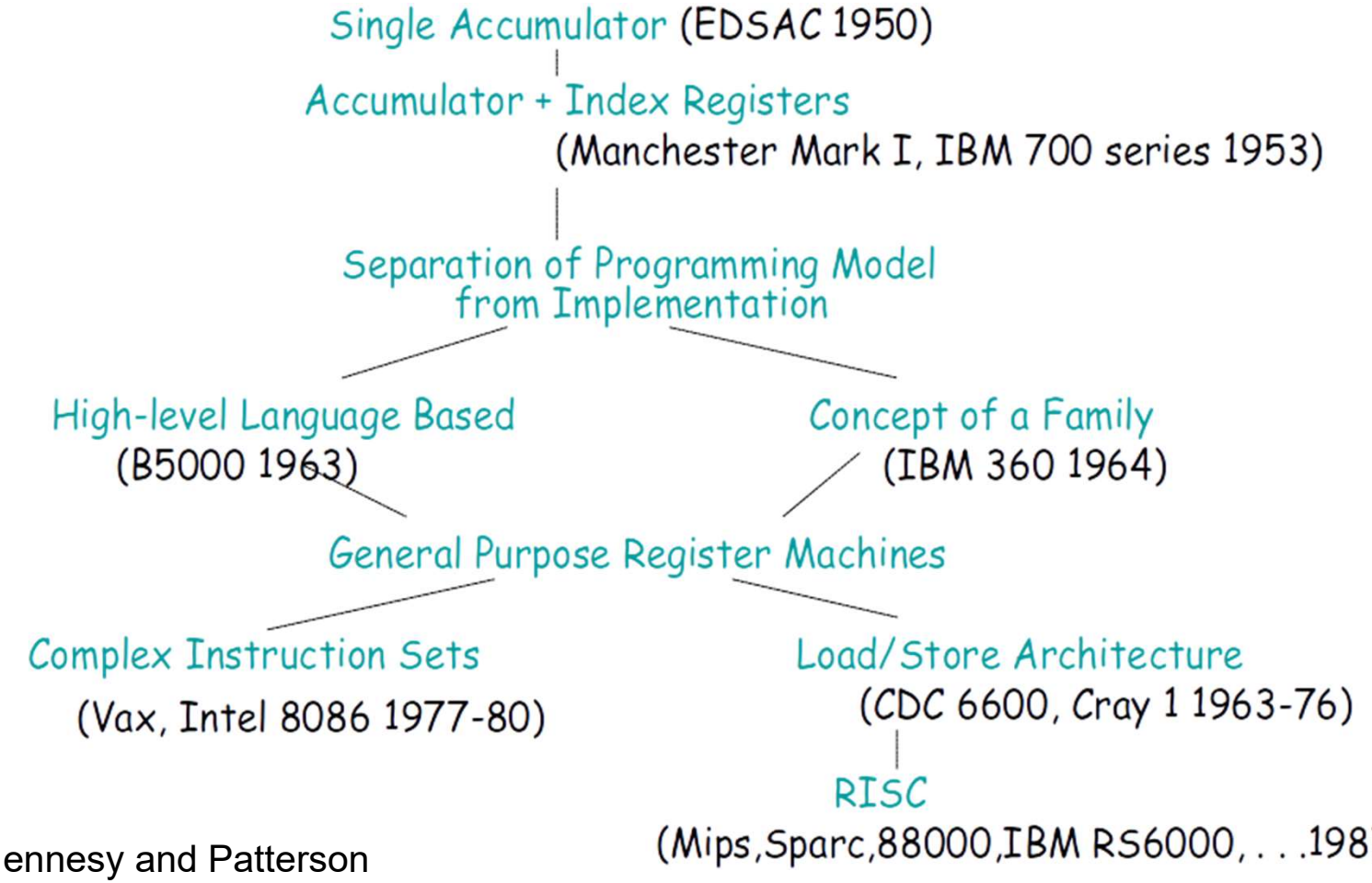
x=(a*b)+(a–(b*c))

```
LD    $r1, a    ; $r1 = a
LD    $r2, b    ; $r2 = b
LD    $r3, c    ; $r3 = c
MUL    $r4, $r1, $r2    ; $r4 = $r1 * $r2 = a*b
MUL    $r5, $r2, $r3    ; $r5 = $r2 * $r3 = b*c
SUB    $r6, $r1, $r5    ; $r6 = $r1 - $r5 = a - (b*c)
ADD    $r7, $r4, $r6    ; $r7 = $r4 + $r6 = (a*b) + (a - (b*c)
SD    $r7, x
```

Register-Register Architecture (ctd.)

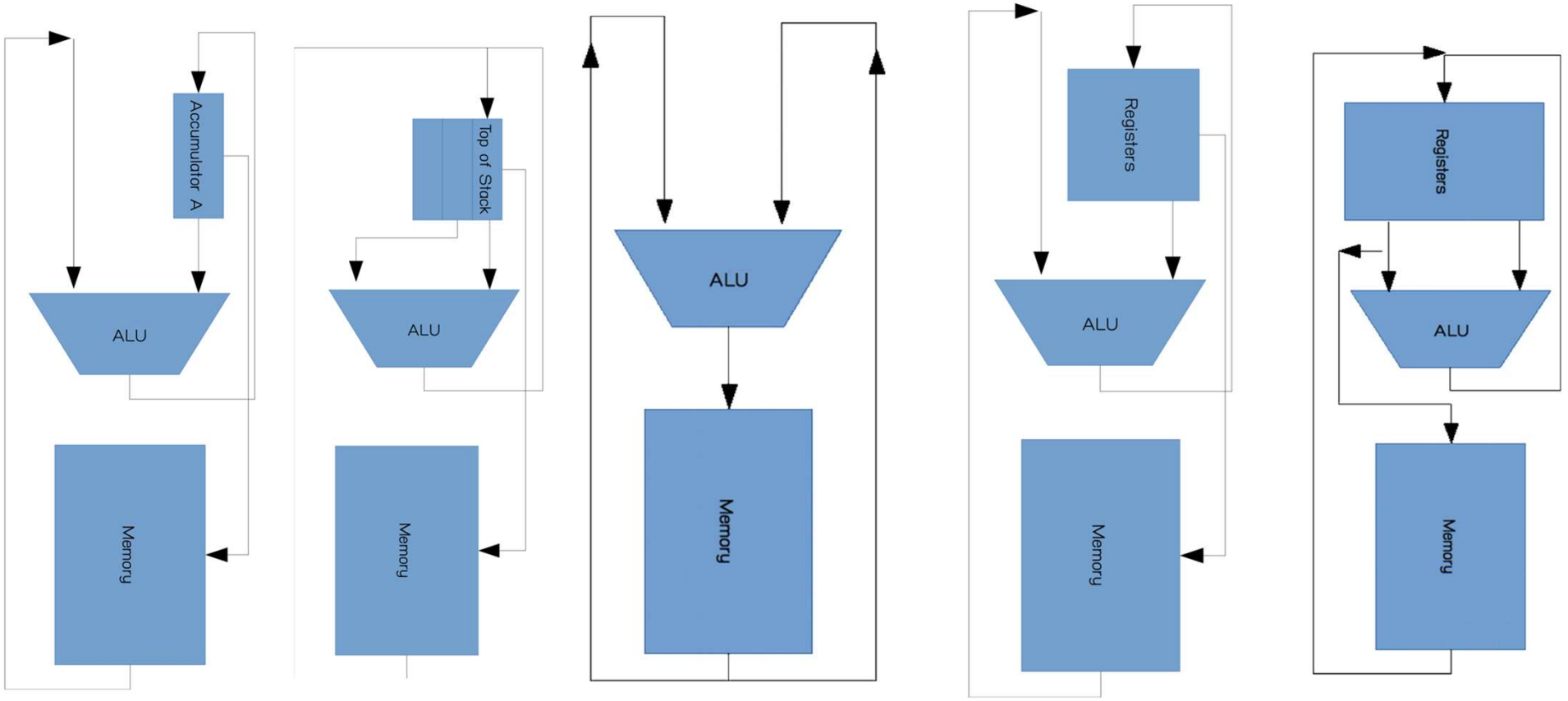
- ★ 8 instructions
- ★ Pros
  - ○ Simple to encode (fixed-length instruction format)
  - ○ Similar CPI
  - ○ Easy to parallelism (including pipelining)
- ★ Cons
  - ○ Higher instruction count
  - ○ Performance depending on good compiler

Evolution of ISA



Taken from Hennesy and Patterson

Comparison of Architecture



Comparison of Architecture (ctd.)

- ★ 2 Paradigms
  - CISC - an instruction is powerful (complex).
  - RISC - an instruction is a basic for building block.
- ★ With the emerging of instruction-level parallelism, RISC wins.
  (We will revisit this in chapter 6.)

- ★ We are in post RISC era. Intel ISA is considered CISC, but it is translated into internal RISC code before performing parallelism.
  The distinction is no longer clear.
- ★ ARM stands for Acorn RISC machine.
  Better at power consumption. (See your mobile devices)

Memory Organization

- ★ Memory is basically a big array indexed by address.
- ★ The smallest unit is a bit. However, we always use byte addressing. To get/set a bit of a byte, use bit mask operation (AND, OR)
- ★ How to store/retrieve multiple-byte data (e.g. short, int, long, ...) ?
  - Ordering (endian)
  - Parallelism (alignment)

Note: In standard C, the *sizeof* function would show the following:

- ★ `sizeof(char) - 1 byte`
- ★ `sizeof(short) - 2 byte(s)`
- ★ `sizeof(int) - 4 byte(s)`
- ★ `sizeof(long) - 8 byte(s)`
- ★ `sizeof(float) - 4 byte(s)`
- ★ `sizeof(double) - 8 byte(s)`
- ★ `sizeof(long double) - 16 byte(s)`

Endian

- ★ Multiple-byte data can simply be stored in a byte addressing memory.
- ★ Base address can be used as a reference. (A variable using 0x0040 and 0x0041 would be referred with 0x0040.)
- ★ How the direction should be stored.
- ★ Little Endian
  - Low-order byte stores at the lowest address
  - Used by Intel
- ★ Big Endian
  - High-order byte stores at the lowest address
  - Used by IBM PowerPC, Sun
- ★ Note that modern processor (ie. ARM from ARM6 onwards) have the option of operating in either little-endian or big-endian mode.

Endian (ctd.)

- ★ In the past, transferring binary data between Mac PowerPC (big endian) and Windows PC (little endian) requires a careful conversion.
- ★ Storing 0x1A2B3C4D

Address	00	01	02	03
Big Endian	1A	2B	3C	4D
Little Endian	4D	3C	2B	1A

- ★ Sometime, programs may store file in a specific endian. (e.g. photoshop store file in big endian regardless of the underlying architecture.)

Endian and performance

- ★ When the size of memory bus is smaller than the actual data, we usually read from lower address to higher address.
- ★ For arithmetic operation, little endian is becoming more natural.
  - Adding the low order byte first allows you to properly handle carry.
- ★ For comparison (more than/less than), big endian is faster.
  - The sign bit is always at the high-order byte

Example. Two 32-bit numbers on 8-bit architecture.

Adding two numbers, little endian allows us to read and to add in order.

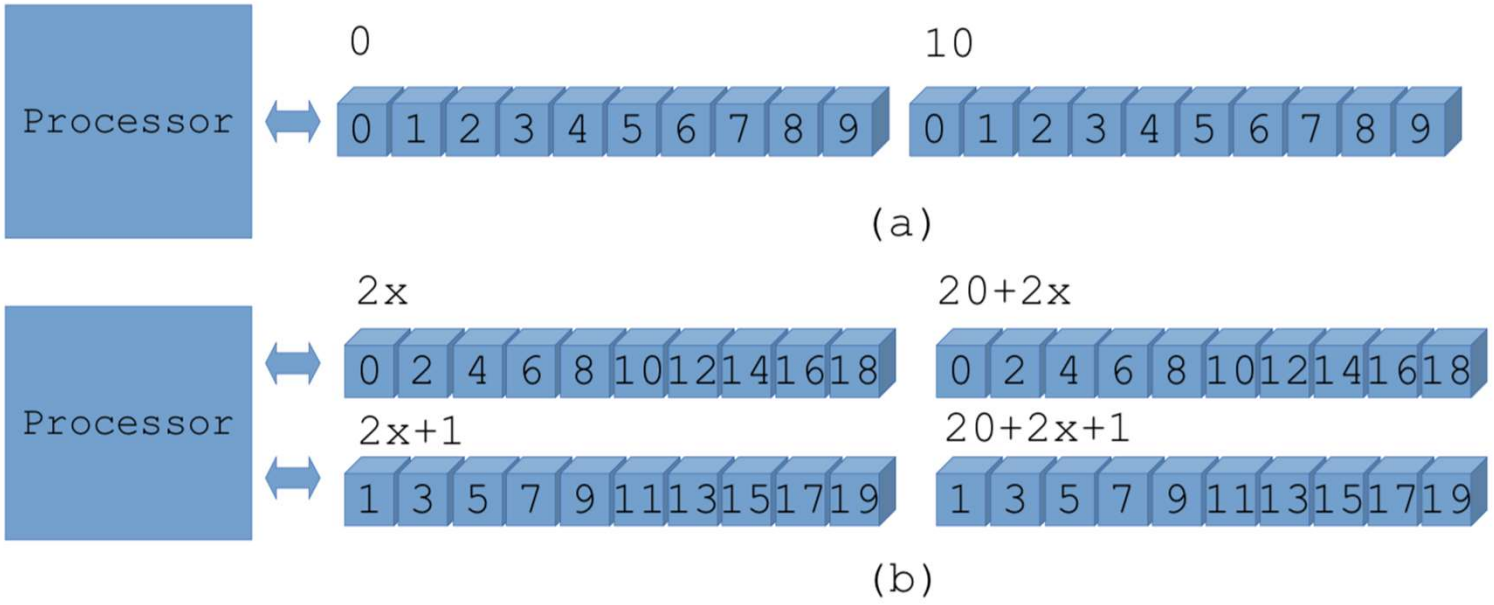
```
0x 12 34 56 78
0x 98 76 54 32

0x AA AA AA AA
```

Comparing two numbers, big endian allows us to answer immediately.

```
0x 12 xx xx xx
0x 98 xx xx xx    ; greater
```

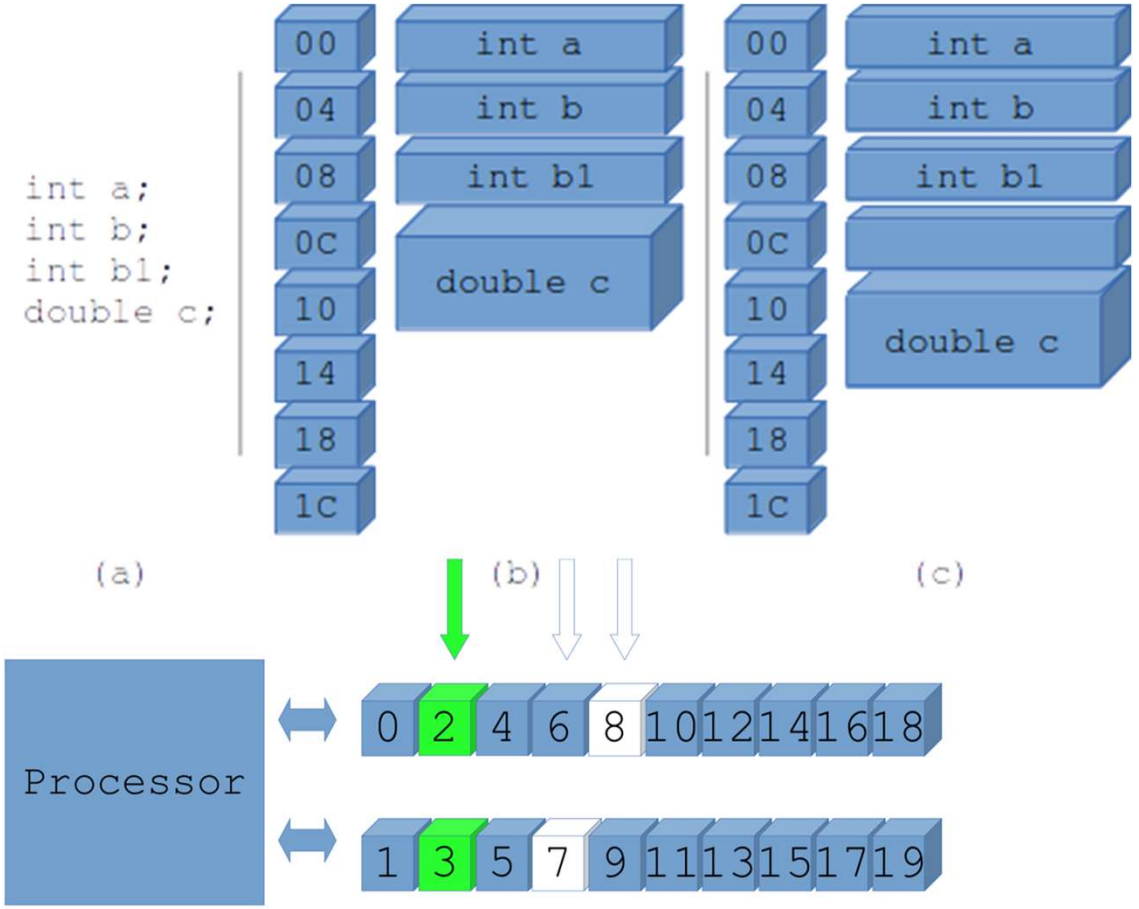
Memory Alignment

- ★ Parallelism in the memory front allows reading multiple bytes at a time.
- ★ If a read gives us two bytes, we can read a short in one cycle.



Restricted vs. Unrestricted Alignment

- ★ Restricted alignment means software must confirm that data can be read at the minimum number of cycles.
- ★ Restricted Alignment means Base Addr. (A) mod size (s) =0



Unrestricted Alignment

- For unrestricted alignment,
  - Software is simple.
  - Hardware must detect and fix the misalignment.
  - Usually take two (or more) memory accesses.
  - Expensive logic?
  - Performance (Slow)?
  - Sometimes required for backwards compatibility with old hardware.

Restricted Alignment

- With restricted alignment,
  - Burden is on the software (compiler).
  - Hardware detect misalignment and traps.
  - No extra time.
- For performance's sake, restricted alignment wins.
- Make the common case fast.

Addressing Mode

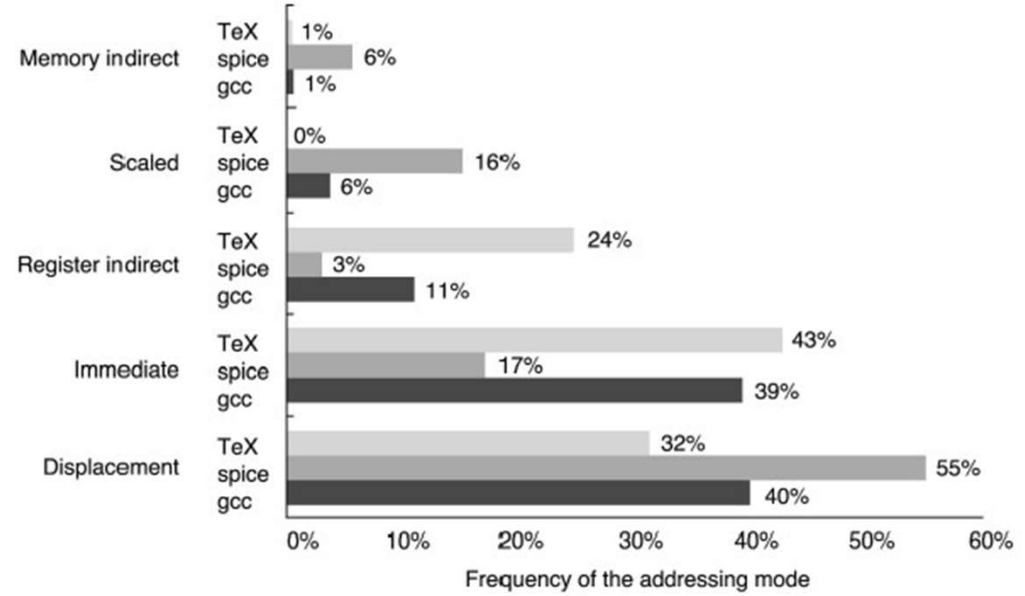
★ Allow a program to easily refer to data in memory

1.	Immediate	LDI $r1, #4	A = 4
2.	Register Direct	ADD $r1, $r2, $r3	A = B + C
3.	Displacement	LD $r1, 100($r2)	A = D[100] หรือ A = G.age
4.	Register Indirect	LD $r1, ($r2)	A = *p
5.	Index	LD $r1, ($r2 + $r3)	A = D[i]
6.	Direct	LD $r1, 1000	
7.	Memory Indirect	LD $r1, @($r2)	A = **p
8.	AutoIncrement	LD $r1, ($r2)+	A = D[i++]
9.	AutoDrecrement	LD $r1, ($r2)-	A = D[i--]
10.	Scale	LD $r1, ($r2)[$r3*4]	F = E[i]

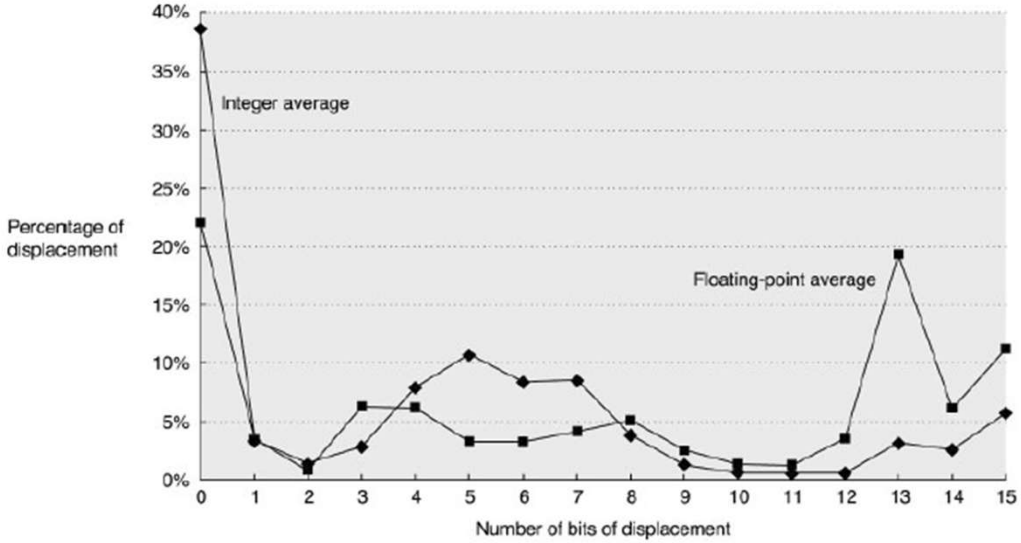
```
unsigned char A, B, C;
char D[200];
int E[200];
int F;
char *p;
struct G {
  char name[100];
  unsigned char age;
}
```

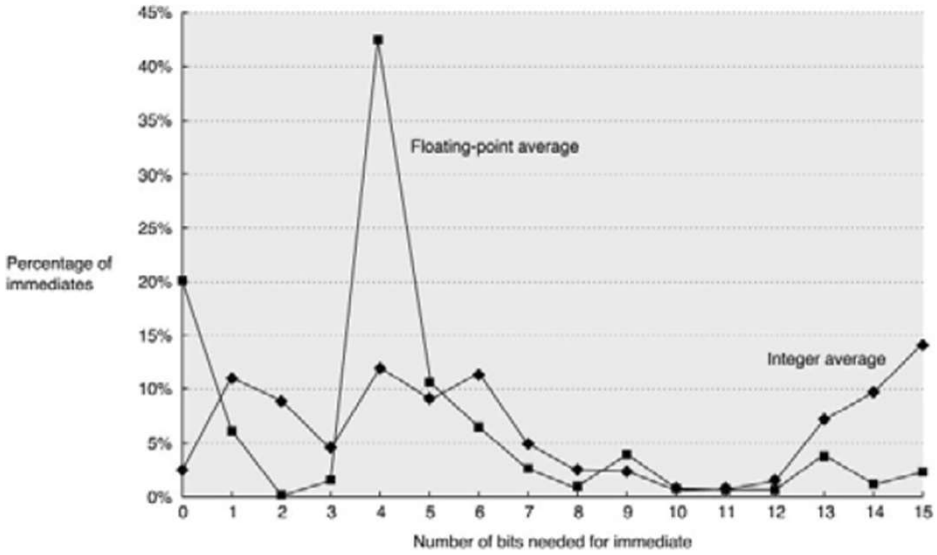
What do we really need?

- ★ Most addressing modes are available in modern architecture.
- ★ Technically, register indirect can replace most modes by first calculating address and put it in a register.



Displacement Value/Immediate Value

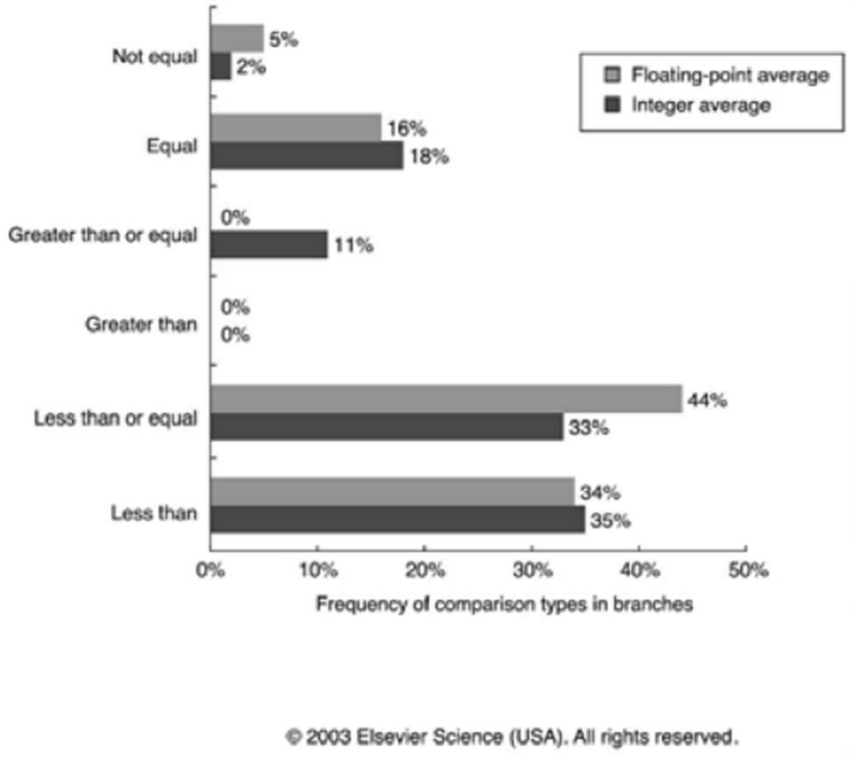


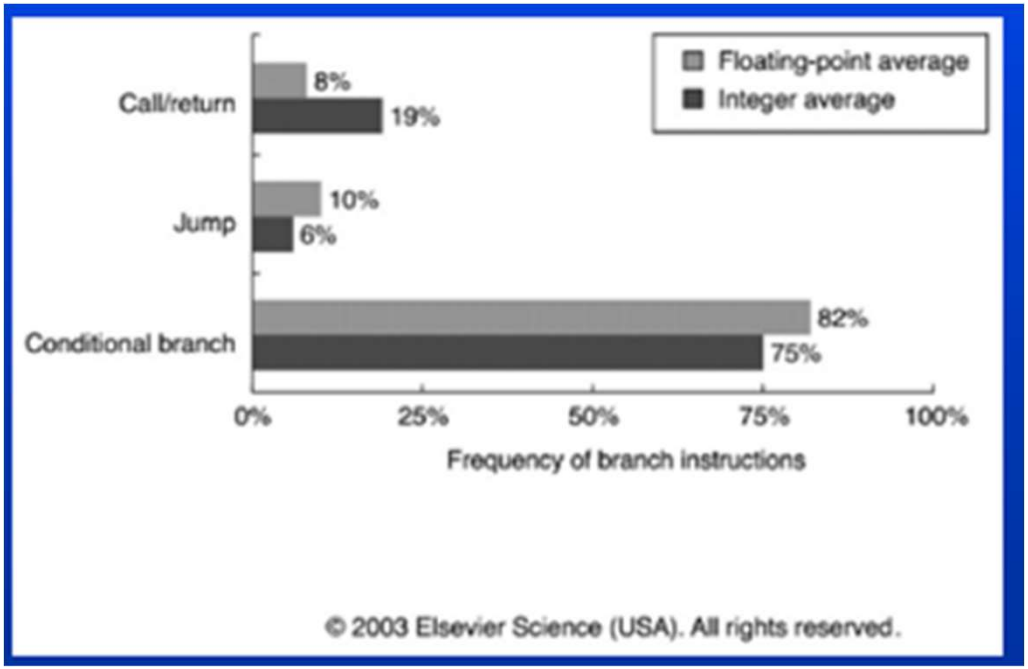


Classes/Types of Instructions

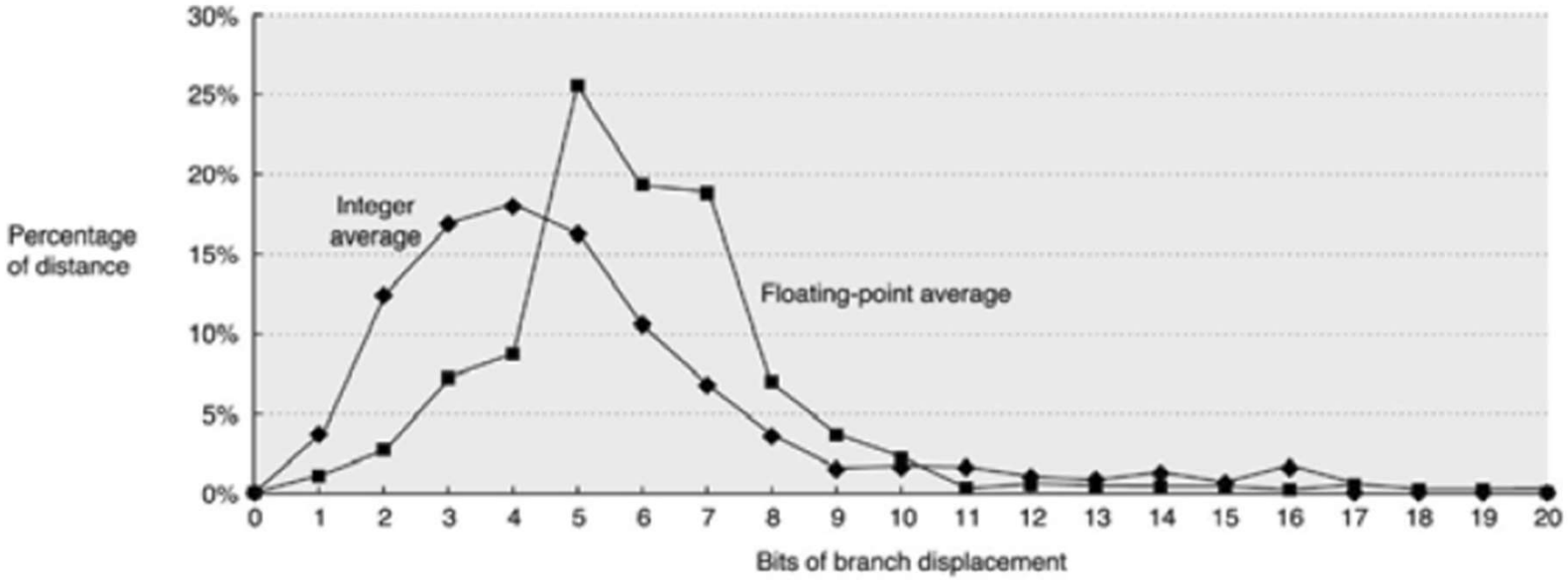
- ★ Data Movement/Transfer Instructions
  - Move data between memory locations, registers, I/O (optional)
  - E.g. LW, SW
- ★ Arithmetics Instructions
  - Mathematical operations between 2 operands
  - E.g. Add, Sub, Mul, Div
- ★ Logic Instructions
  - Logical operations
  - E.g. And, Or, Xor, Com (Not)
- ★ Control Instructions
  - Change control flow (PC <- [xxxx]. Normally, PC<- PC+1)
  - E.g. JMP, BEQ
- ★ Extended Instructions (Multimedia, Encryption)

Control Flow





Jump Distances



Addressing Modes in Control Flow

- ★ Usually displacement is relative to PC (why?)
- ★ For Some addresses, we do not known statically (eg. shared library).
  - Use register containing address.
  - (In really, there is a symbol table for this.)
- ★ Relocatable Program
- ★ Dynamic binding

```
Loop:
          ....
     ;BEQ loop
     BEQ -120


;CALL _printf
CALL $r14
```

Trends in Addressing Mode

- ★ For a new ISA design,
  - Support for displacement, immediate, register indirect addressing modes
  - 12-16 bits of displacement field
  - 8-16 bits of immediate field
- ★ For complex addressing mode, let the compilers transform it to simple addressing mode.
- ★ Hardware only focuses on simpler modes.

Sub Routine

- ★ A call to a subroutine means
  - **Passing parameter**
  - **Save current status**
  - Save the return address
  - Call subroutine (Change PC to the address subroutine)
  - …… (do the subroutine work)
  - …… Save return data
- ★ A return from a subroutine means
  - Restore status
  - Load return address to PC
- ★ Usually defined in application binary interface (ABI)

Parameter Passing

- ★ Register
- ★ Stack Frame

```
int min(a,b) {
          if(a>b) {
                    return b;
          }
          return a;
}

min(5,4);
```

```
_min:
          CMP       $r3, $r4
          BLE       _less
          MOV       $r1, $r4
          RET
_less:
          MOV       $r1, $r3
          RET


     ...
     LI   $r3, #5
     LI   $r4, #4
     CALL _min
```

```
_min:
          LD        $r3, [$r14+4]
          LD        $r4, [$r14+8]
          BLE       _less
          MOV       $r1, $r4
          RET
_less:
          MOV       $r1, $r3
          RET


     ...
     PUSH #5
     PUSH #4
     CALL _min
```

Assuming that $r14 is stack

Caller Save vs. Callee Save

- ★ Assume that a routine is using $r1.
- ★ In reality, caller do not know which registers are used by the routine.
  - Intel x86 provides PUSHA and POPA to save all registers. (Simpler, but slower)
- ★ Callee save has more benefit?

Caller Save

```
ADD     $r1, $r1, $r2
PUSH    $r1

CALL    SUB

POP     $r1
ADD     $r2, $r2, $r3
        ...

SUB:


ANI     $r1, $r1, #00
ADD     $r1, $r7, $r8


RET
```

Callee Save

```
ADD     $r1, $r1, $r2


CALL     SUB


ADD     $r2, $r2, $r3
    ...

SUB:
PUSH      $r1

ANI     $r1, $r1, #00
ADD     $r1, $r7, $r8

POP       $r1
RET
```

Exceptions

★ What is an exception?

- ★ Unprogrammed Control Flow
- ★ Classified by sources
  - Hardware/External - Interrupt
  - Software - trap
  - Intel uses interrupt for both hardware and software exceptions
- ★ When an exception occurs,
  - Save status
  - Save current instruction (PC)
  - Call the interrupt handles
  - … (do the handler routine)
  - Restore status
  - Restore saved address (PC)

The mechanism is similar to that of the subroutine.

Where is the handlers?

- ★ Vector Table
  - Address in the table
- ★ Handler Table
  - Code in the table
- ★ Fixed Entry
  - Let the software check status and make a decision

Vector Table

- ★ Address in the table
- ★ Address =
  Base +
  (Source no. * address size)
- ★ Assume 32-bit address,
  interrupt 8
- ★ Address=mem[0x8000+(8*4)]
- ★ JUMP to address

+56	0x……
+52	0x…….
…	
+8	0x00004000
+4	
+0	0x88000004

Handler Base
0x8000

Handler Table

- ★ Code in the table
- ★ Handler =
  Base + (Source no. * handler size)
- ★ Assume 16 bytes handler size,
  interrupt 8
- ★ Address=0x8000+(8*16)
- ★ JUMP Address

+240	IRET
+224	IRET
...	
+32	MOVE $..... .. IRET
+16	
+0	IRET

Handler Base
0x8000

Fixed Entry

- ★ Regardless of the interrupt source, CPU always jump to 0x1000.
- ★ A program check interrupt source register and make a decision from that.

Instruction Format

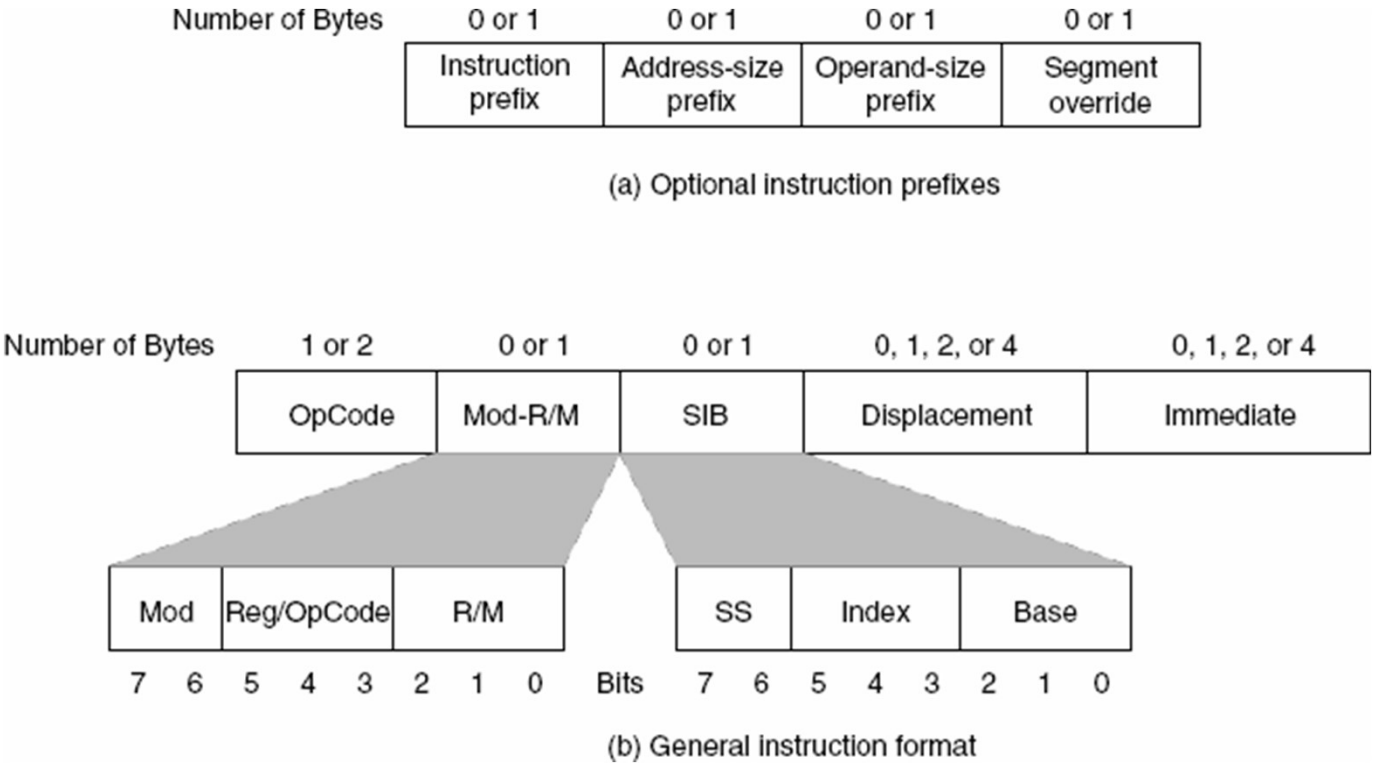
- ★ Encoding of instructions affects:
  - Size of code
  - Decoding work (of the processor)
- ★ Depends on:
  - Range of addressing modes
  - Degree of independence between opcodes and modes
- ★ We want:
  - As many registers and addressing modes as possible
  - Reduce the size of instructions (and programs)
  - Ease of decoding and implementing in hardware.

Three (Basic) Variations

- ★ Variable instruction length
  - Bits in the first fetch tell the length
  - Use address specifier
  - Support several modes and operations
  - Smaller code?
- ★ Fixed instruction length
  - Small number of addressing mode
  - Opcode imply addressing mode
  - Larger code?
- ★ Hybrid approach
  - Arm thumbs support java byte code on chip (at higher CPI)

Variable Instruction Length

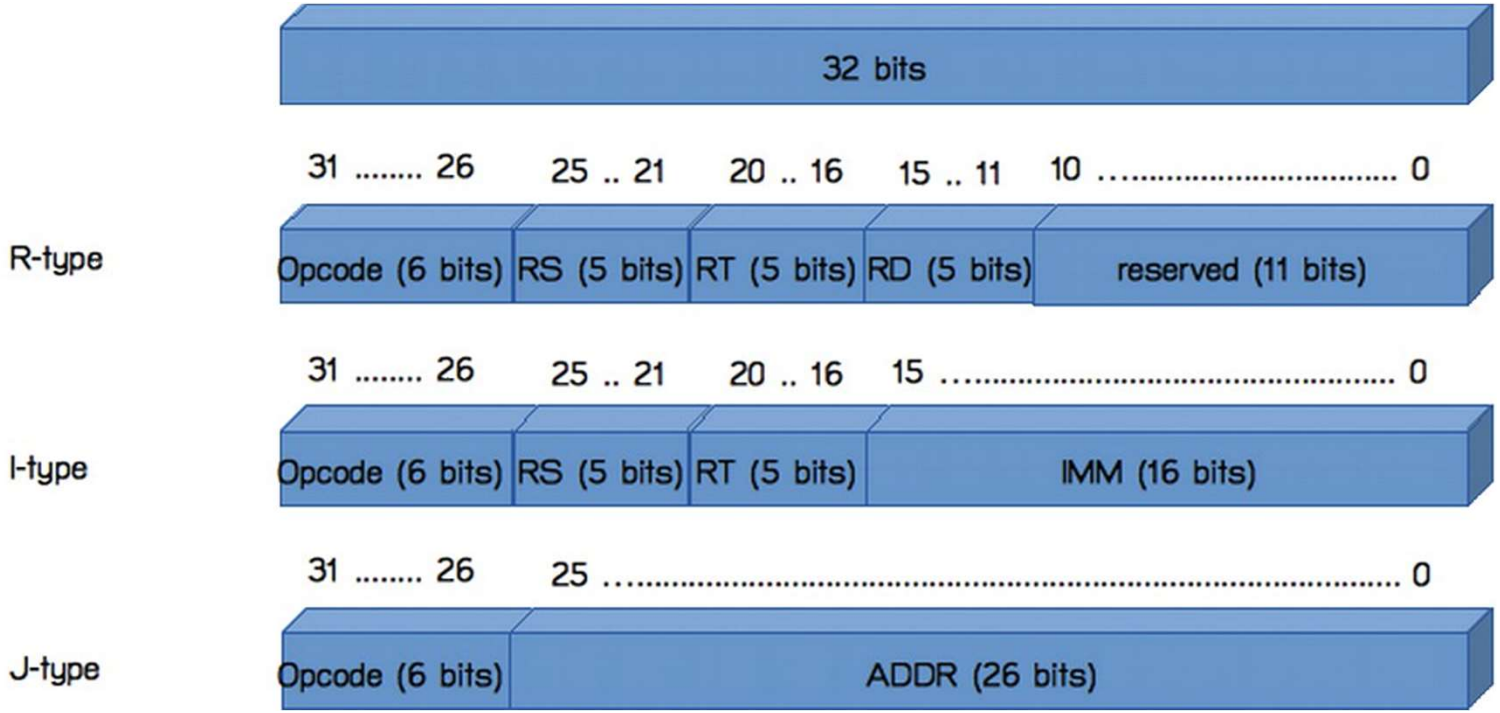
- A sample from Intel x86



Taken from http://www.c-jump.com/CIS77/CPU/x86/lecture.html

Fixed Instruction Length

★ A sample from nano LADA

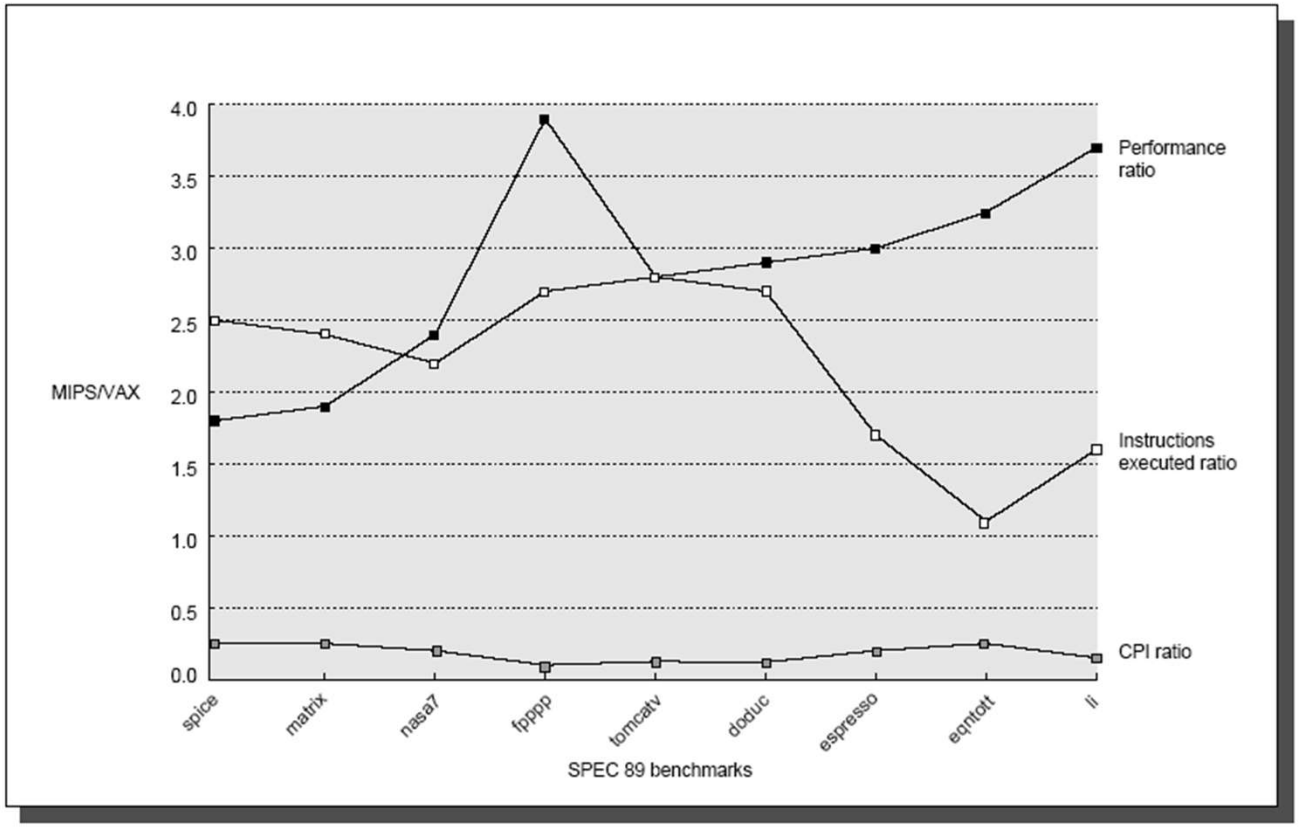


Trends

- ★ For performance, fixed Instruction Length wins.
- ★ However, we have previous design. (We will revisit this in PostRISC.)

MIPS code is usually twice the size of VAX code.
MIPS CPI is ⅙ of VAX CPI.

Which one is faster?

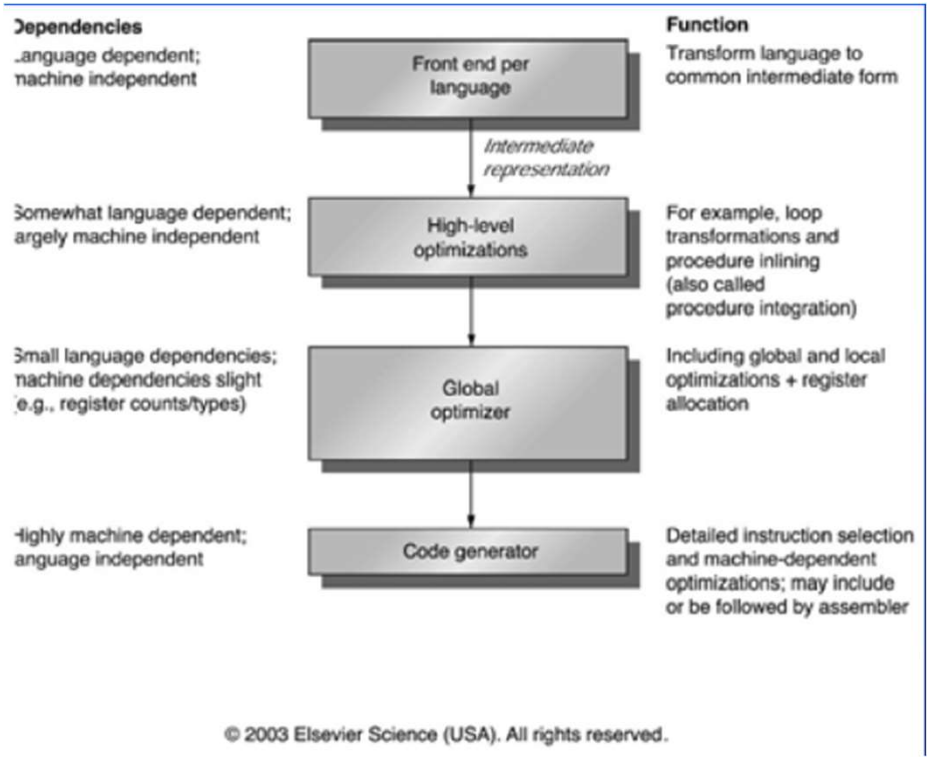


Roles of Compiler in Optimization

- ★ Instruction set architecture is the target of compiler.
- ★ Instruction set architecture should allow compilers to generate efficient code.
- ★ Goals of Compiler
  - ○ Correctness
  - ○ Speed (of code)
  - ○ Compile time
  - ○ Debugging support

Compilation Steps

- ★ Compile Language specific to intermediate code
- ★ High-level Optimization
- ★ Global Optimization
- ★ Code generator



Optimizations

- ★ High-level
  - Procedure integration (procedure in-lining)
- ★ Local (within a basic block)
  - Common expression elimination
  - Constant propagation
  - Stack height reduction
- ★ Global (beyond a basic block)
  - Global common expression elimination
  - Copy propagation
  - Code motion
- ★ Machine-Dependent
  - Strength reduction (shift for multiplication)
  - Branch offset optimization
  - Pipeline scheduling

Note: A basic block is a single sequence of code with no jump or branch.

Examples of Optimizations

Copy propagation

```
// before
m = 60;
h = 60 * m;
d = 24 * h;
days = seconds / d;
```

```
// after
days = seconds / 86400;
```

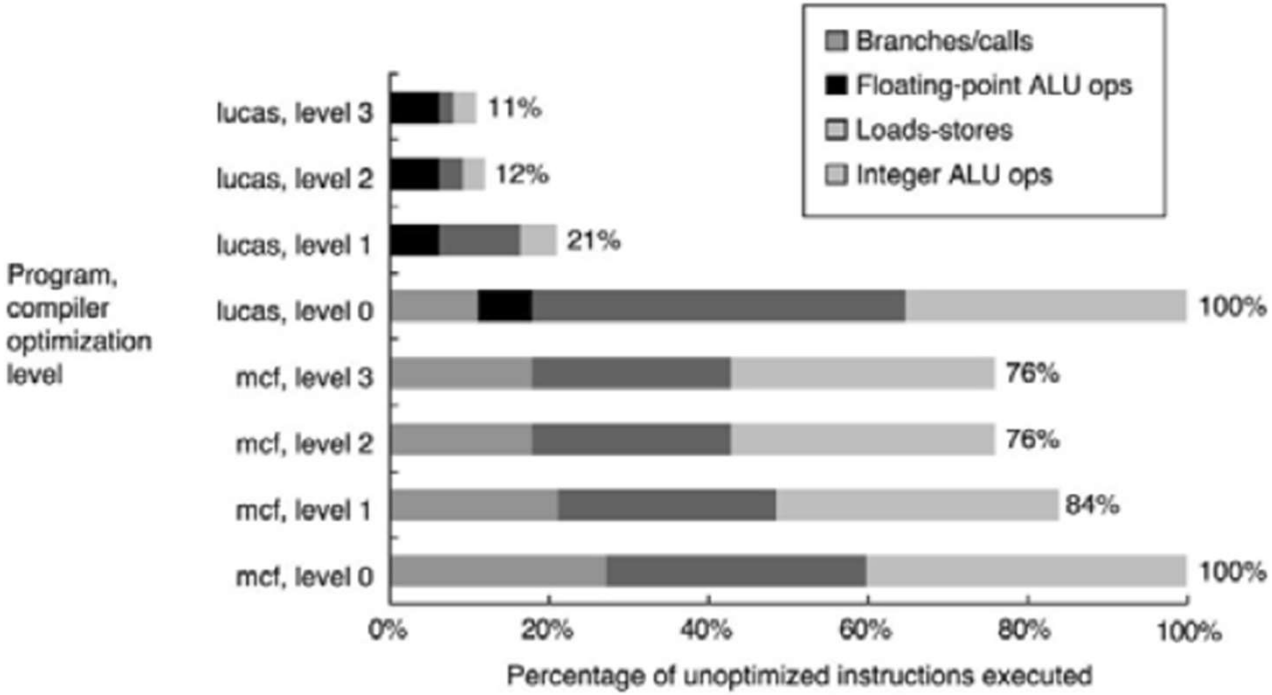
Common-expression elimination

```
// before
a = b + c + d;
y = b + c + e;
```

```
// after
x = b + c;
a = x + d;
y = x + e;
```

Effectiveness of Optimization

- ★ Programs are locally simple, but globally complex.
- ★ Most bugs in compilers are from optimizations.
- ★ Alias (Pointer) prevents allocating registers to variables



https://gcc.gnu.org/bugzilla/buglist.cgi?component=c%2B%2B&order=changeddate%20DESC%2Cbug_status%2Cpriority%2Cassigned_to%2Cbug_id&product=gcc&query_format=advanced&resolution=---

https://en.wikipedia.org/wiki/Undefined_behavior

How to help compiler writer

- ★ Make the common case fast?
- ★ Desirable ISA properties (from compiler's point of view)
  - Regularity
    - Orthogonality among addressing mode, data types, and operations
  - Simplicity
    - Provide primitives, not solutions
  - Simplify tradeoffs
  - Allow compile-time binding

Exercises

Which one is the fastest?

- ★ Hardware-specific optimization.
- ★ Let's clear a register
  - CLR R1
  - MOV R1,0
  - SUB R1, R1, R1
  - XOR R1, R1, R1
- Which one is the fastest?
- ★ Is it funny?

For most architecture, the recommended method is XOR.

Optimize the following code

★ Optimize the following code:

```
for(x=0;x<=100;x++) {
        hs=3600;
        ds=hs*24;
        h=a[x]/hs;
        d=a[x]/ds
}
```

Optimize the following code

- Consider this three statements:
  - A=B+C;
  - B=A+C;
  - D=A-B;
- Use the technique of copy propagation to transform the code sequence to the point where no operand is a computed value. Note the instances in which the transformation has reduced the computational work of a statement and those cases where the work has increased. What does this suggest about the technical challenge faced in trying to satisfy the desire for optimizing compilers?

Memory Allocation

- With restricted alignment, draw a diagram showing the memory allocation for following variables
  - char a;
  - short b;
  - int c;
  - char d;
  - long e;
  - double f;

Assignment 1

For more information, see the announcement.

End of Chapter 3