Lecture 6

Computer Architecture
(Spring 2020)

Instruction Set Principles

Dr. Duo Liu (刘铎)
Office: Main Building 0626
Email: liuduo@cqu.edu.cn

Early 1960s: IBM Starts Talking of “Computer Architecture”

“the term architecture is used here to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls, the logic design, and the physical implementation.”

M. Amdahl, G. A. Blaauw, and J. F. P. Brooks. "Architecture of the IBM System/360," IBM Journal of Research and Development, 1964.

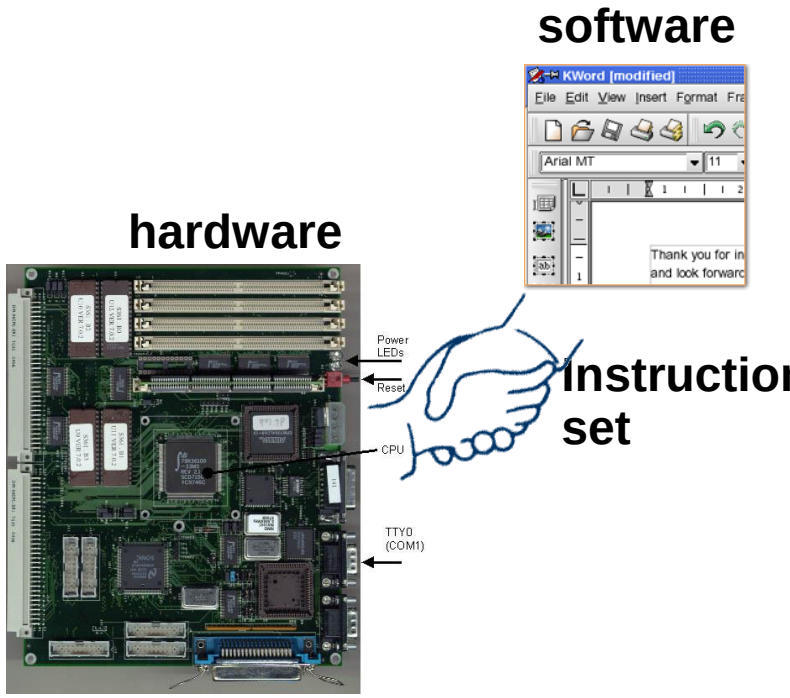
- Early 1960s: IBM launches the IBM 360 , a landmark ISA
  - IBM coins the term computer architecture to refer to the program-visible portion of the instruction set
    - today the term has a broader meaning (design the whole computer system)
  - a family of computers that are able to run the same software
    - this “family” concept (a platform in today’s jargon) was quite novel at the time (IBM had 5 different architectures before the 360)
  - the first successful computer with a general-purpose register (GPR) organization

Shift in Applications Area

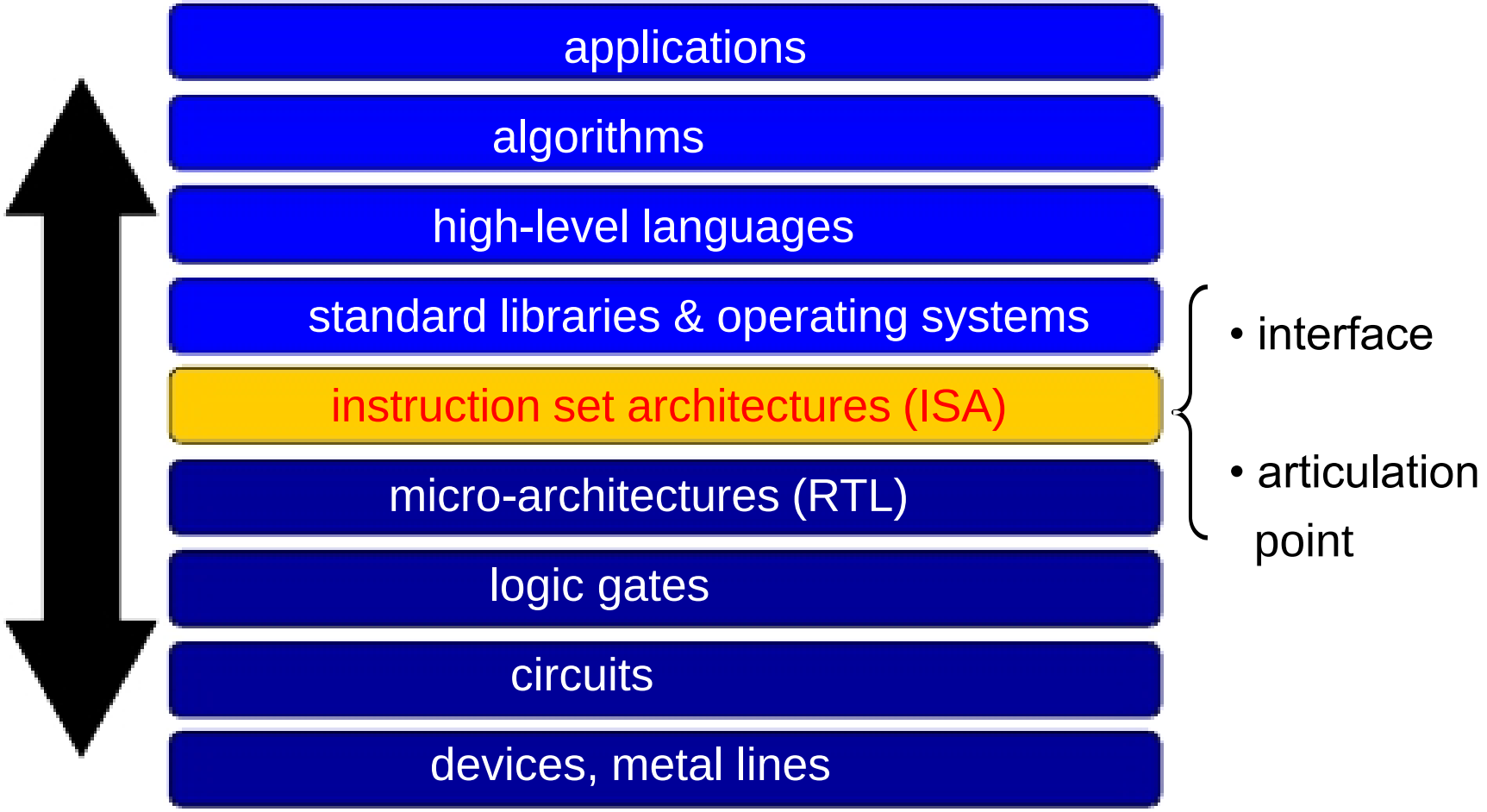
- Desktop Computing – emphasizes performance of programs with integer and floating point data types;
  little regard for program size or processor power
- Servers - used primarily for database, file server, and web applications; FP performance is much less important for performance than integers and 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

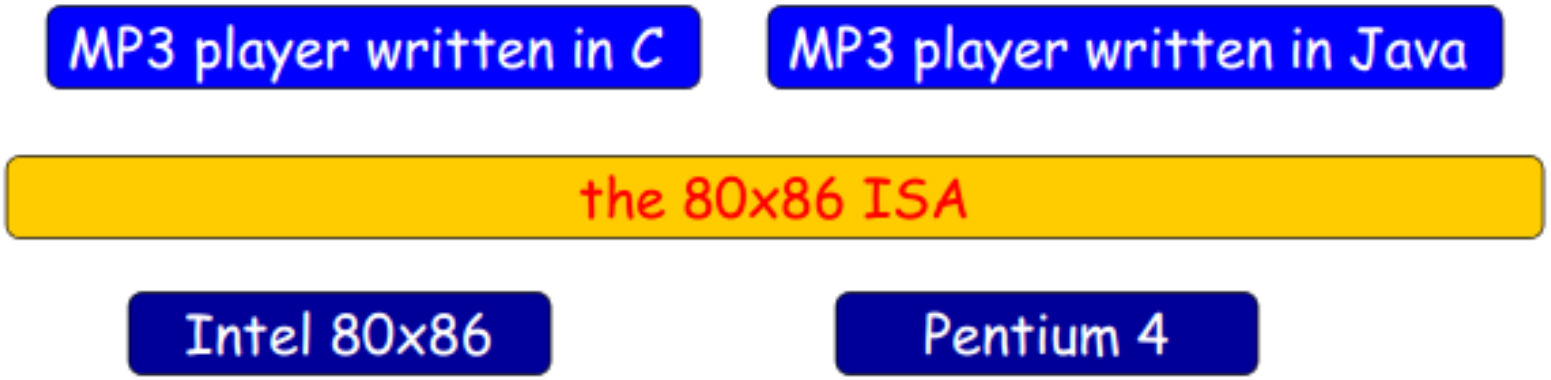


Computer Architecture and Abstraction Layers



ISA: Articulation Point between HW & SW

- Articulation point of the design process
  - decouples implementation and specification
    - the HW implementation of the microprocessor is decoupled from the specification of the instruction set
  - decouples high-level and programs languages from the abstract machine
    - through the role played by the compiler/interpreter



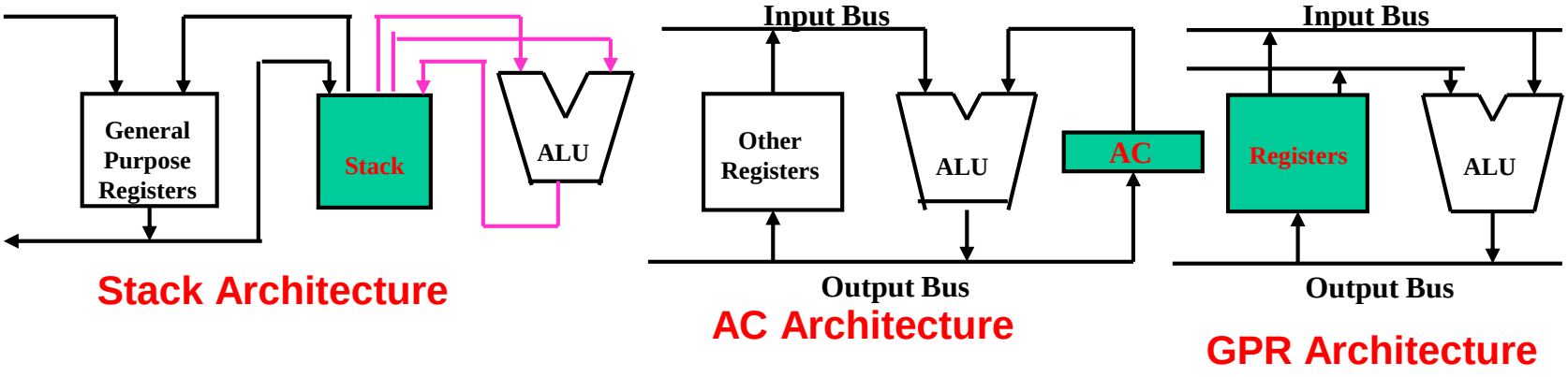
ISA: Interface between HW & SW

- Abstraction layer
  - that hides the details of the underlying levels while exposing the key information to optimize the design
    - state of the microprocessor (registers, memory, PC)
    - HW-supported instructions to operate on this state
  - theoretically limits the space of design exploration, yet still enables good results in a more predictable, and often shorter time
    - high-level languages + compiler optimizations vs. manual optimization of hand-written assembly code
  - enables the completion of more complex projects by simplifying the validation phase

Taxonomy of ISAs

Computer Architectures are classified into three classes according to the Register Structures for operands storage

- **Stack Architecture:** operands are implicitly on the top of the stack
- **AC Architecture:** one operand is implicitly accumulator
- **General Purpose Register Computer Architecture**
  - 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



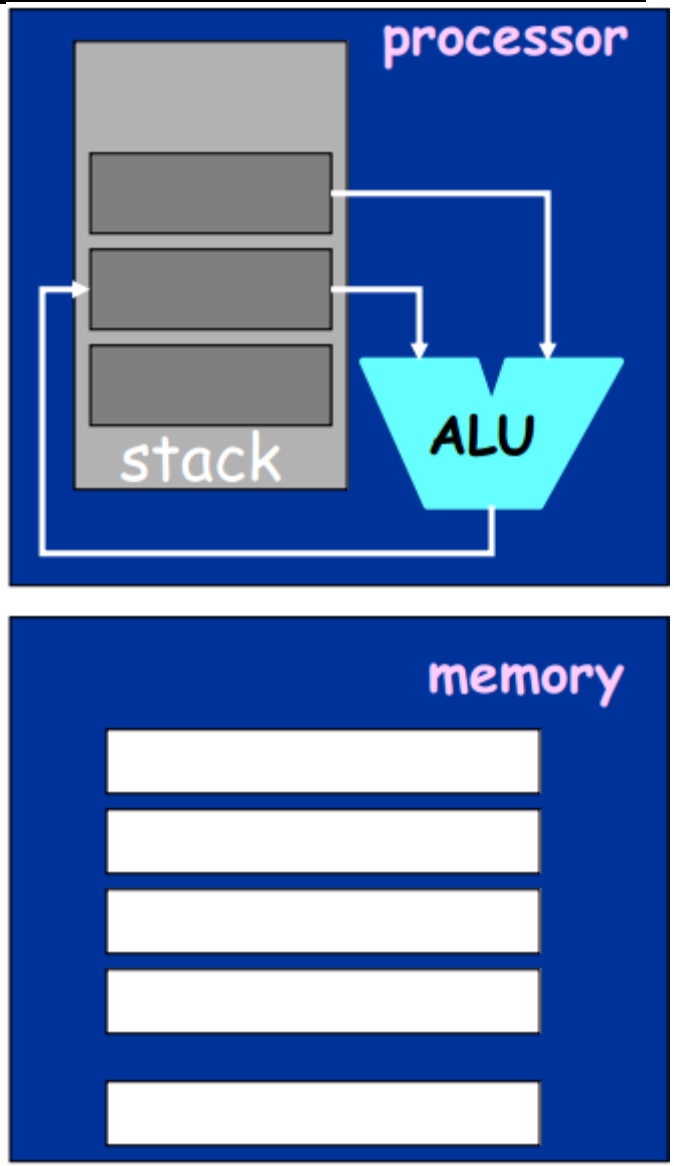
Taxonomy of ISAs: Stack

- Instruction operands
  - none (implicit) for ALU operations
  - one for transfer from/to memory
    - push/pop

a = b + (c * d)

```
push b
push c
push d
mul
add
pop a
```

- Pros
  - short instruction
  - simple compiler
- Cons
  - inefficient code
    - many swaps, copies
  - stack may be slow
- Examples
  - B5000, JVM



Stack Machines

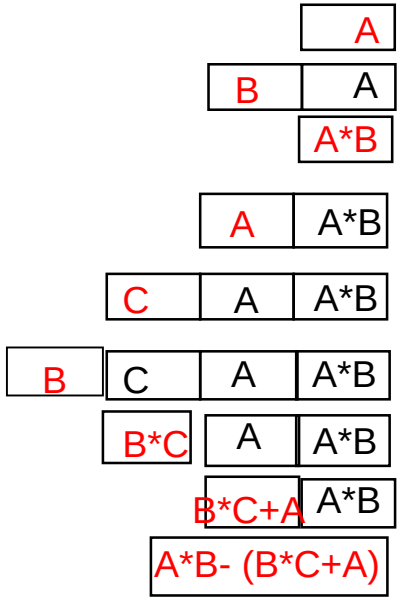
- Instruction set:

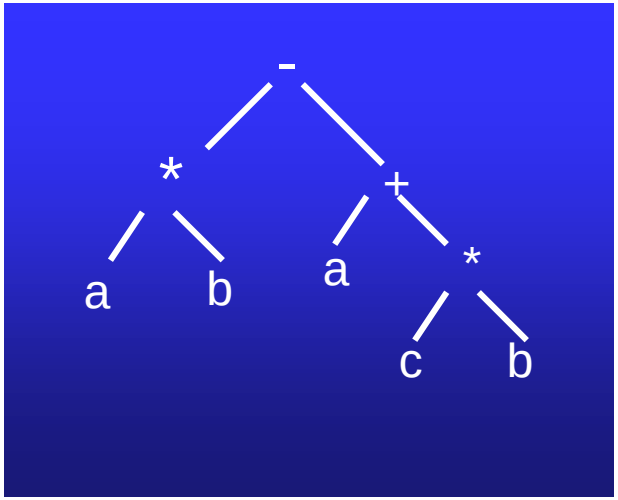
  Arithmetic operators(+, -, *, /, . . .)

  push A, pop A

- Example: a*b - (a+c*b) ⟶ ab*(a(cb)*+)-

```
      push a
push b
 *
push a
push c
push b
 *
 +
      -
```



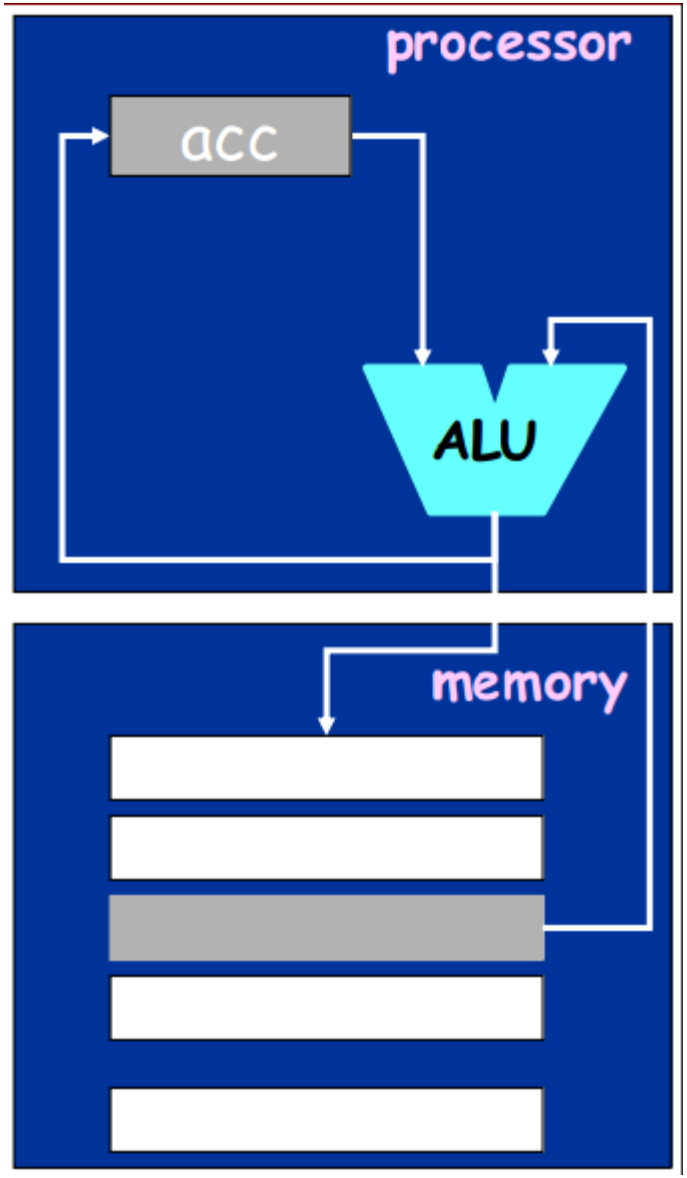


Taxonomy of ISAs: Accumulator

- Instruction operands
  - 1 explicit, 1 implicit
    - acc←acc + *mem
    - acc ← *mem
    - *mem ← acc
- Pros
  - short instruction
  - simple design
- Cons
  - inefficient code
    - many transfers
  - pipelining is hard
- Examples
  - Early machines (EDSAC, IAS)

a = b + (c * d)

```
load c
mul d
add b
store a
```

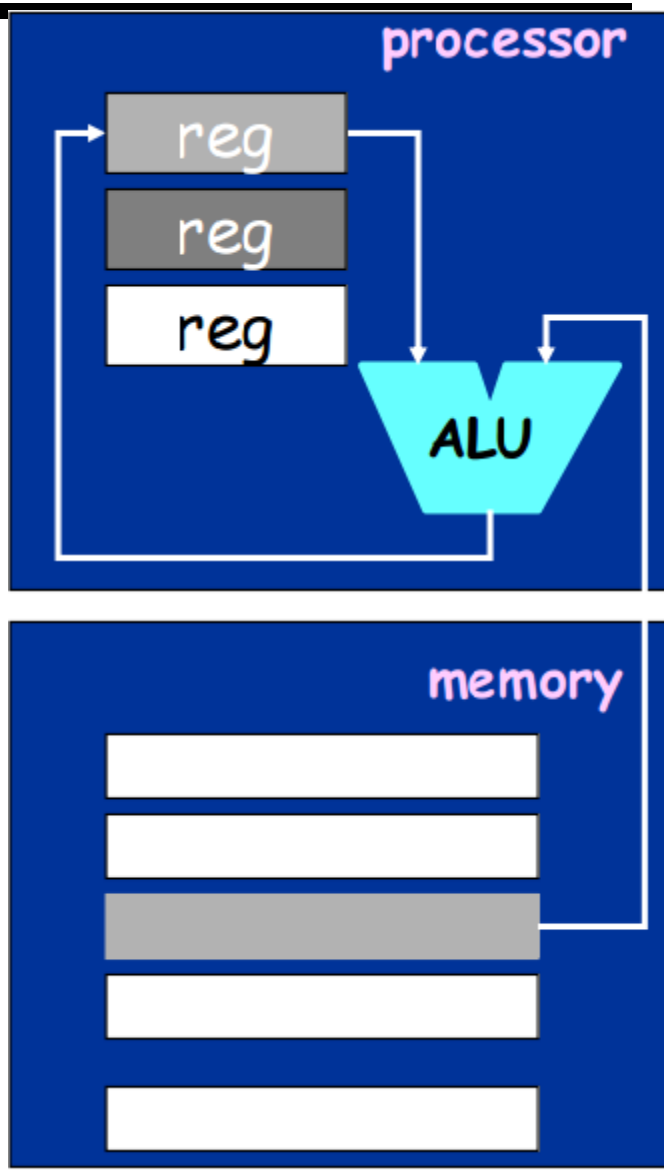


Taxonomy of ISAs: Register-Memory

- Instruction operands
  - 2 (typically)
    - one from memory
- Pros
  - fewer instructions
  - dense encoding
- Cons
  - operand asymmetry
    - result destroys one
  - pipelining is hard
    - different CPIs
- Examples
  - IBM 360, 80x86, Motorola 68000, TI

a = b + (c * d)

```
load r1, c
mul r1, r1, d
add r1, r1, b
store r1, a
```

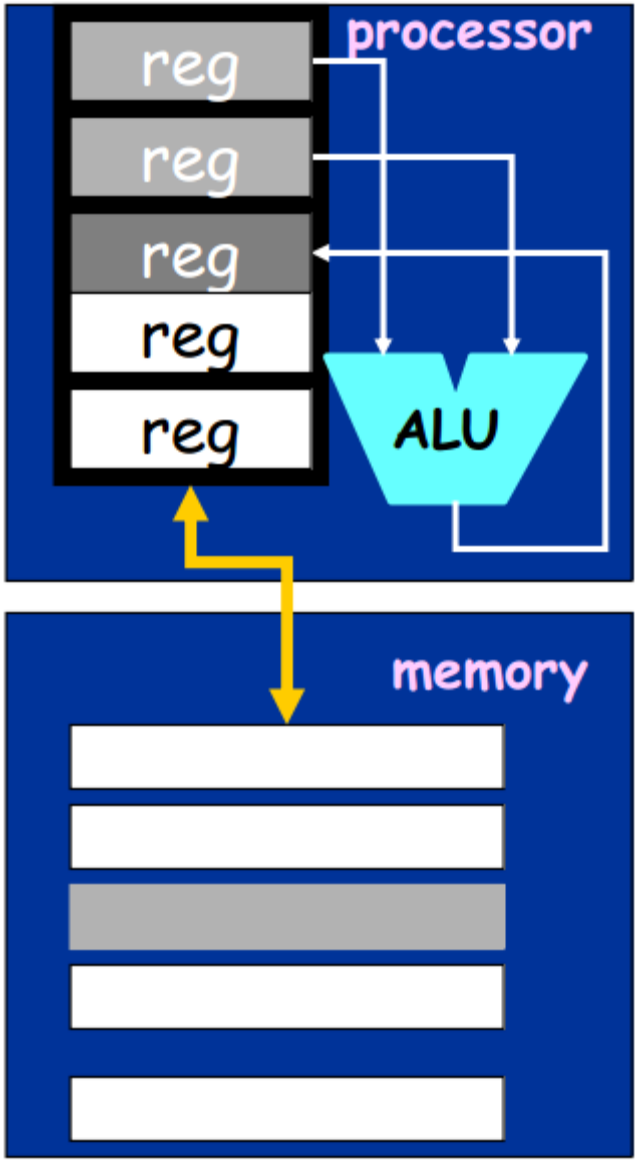


Taxonomy of ISAs: Register-Register (or Load-Store)

- Instruction operands
  - 3 (typically)
    - from registers
- Pros
  - simple, symmetric
    - faster instructions
    - smarter compilation
- Cons
  - higher instr. count
  - longer encoding
  - lower density
- Examples
  - CDC6600, CRAY-1, Alpha, MIPS, SPARC, PowerPC

a = b + (c * d)

```
load r1, c
load r2, d
load r3, b
mul r4, r1, r2
add r5, r4, r3
store r5, a
```

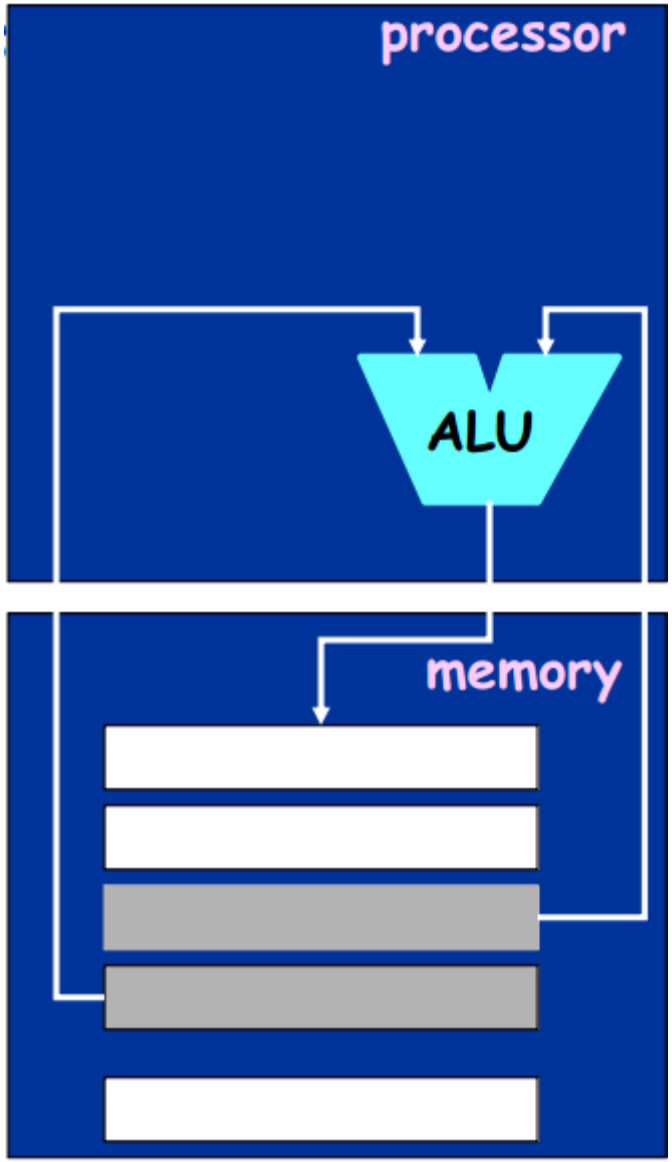


Taxonomy of ISAs: Memory-Memory

- Instruction operands
  - 2 or 3 operands
    - all from memory
- Pros
  - most compact instruction count
  - no need of registers
- Cons
  - large variation in instruction lengths
    - result destroys one
  - pipelining is hard
    - very different CPIs
- Examples
  - VAX (some instr.)
  - not used nowadays

a = b + (c * d)

```
mul e, c, d
add a, e, b
```



GPR Machines

Type	Advantages	Disadvantages
Register-register (0,3)	Simple, fixed-length instr. encoding. Simple code generation model	Higher instruction count. 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.	A source operand is destroyed. Clocks per instruction varies by operand location.
Memory-memory (3,3)	Program becomes most compact. No waste of registers for temporaries.	Large variation in instruction sizes and in work per instruction. Memory accesses create memory bottleneck.

Example

```
(0,3): ADD   R1,R2,R3      R[R1]← R[R2] + R[R3]
(1,2): ADD   R1, X         R[R1]← R[R1] + M[X]
(3,3): ADD   X1,X2,X3      M[X1]← M[X2] + M[X3]
```

GPR Machines

- **Maximum number of operands(O)**
  - **two or three operands**
- **Number of memory addresses(M)**
  - **0,1,2,3**

Type (M,O)	No of memory addresses	Maximum No of operands allowed	Examples
(0,3)	0	3	SPARC, MIPS, PowerPC, ALPHA
(1,2)	1	2	Intel 80x86, Motorola 68000
(2,2)	2	2	VAX
(3,3)	3	3	VAX

R-R vs RM

A+B+C

RR Instructions

```
LD      R1,A
LD      R2,B
LD      R3,C
ADD     R4,R1,R2
ADD     R5,R4,R3
```

RM instructions

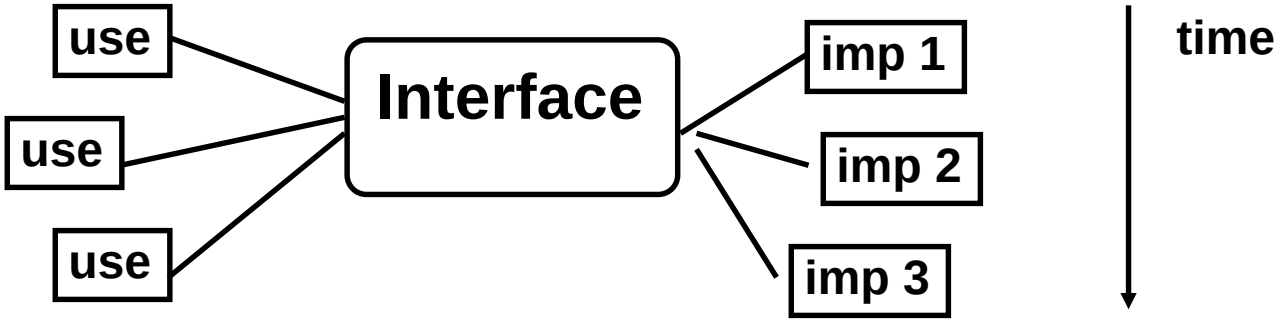
```
LD      R1,A
ADD     R1,B
ADD     R1,C
```

RM instructions reduce IC

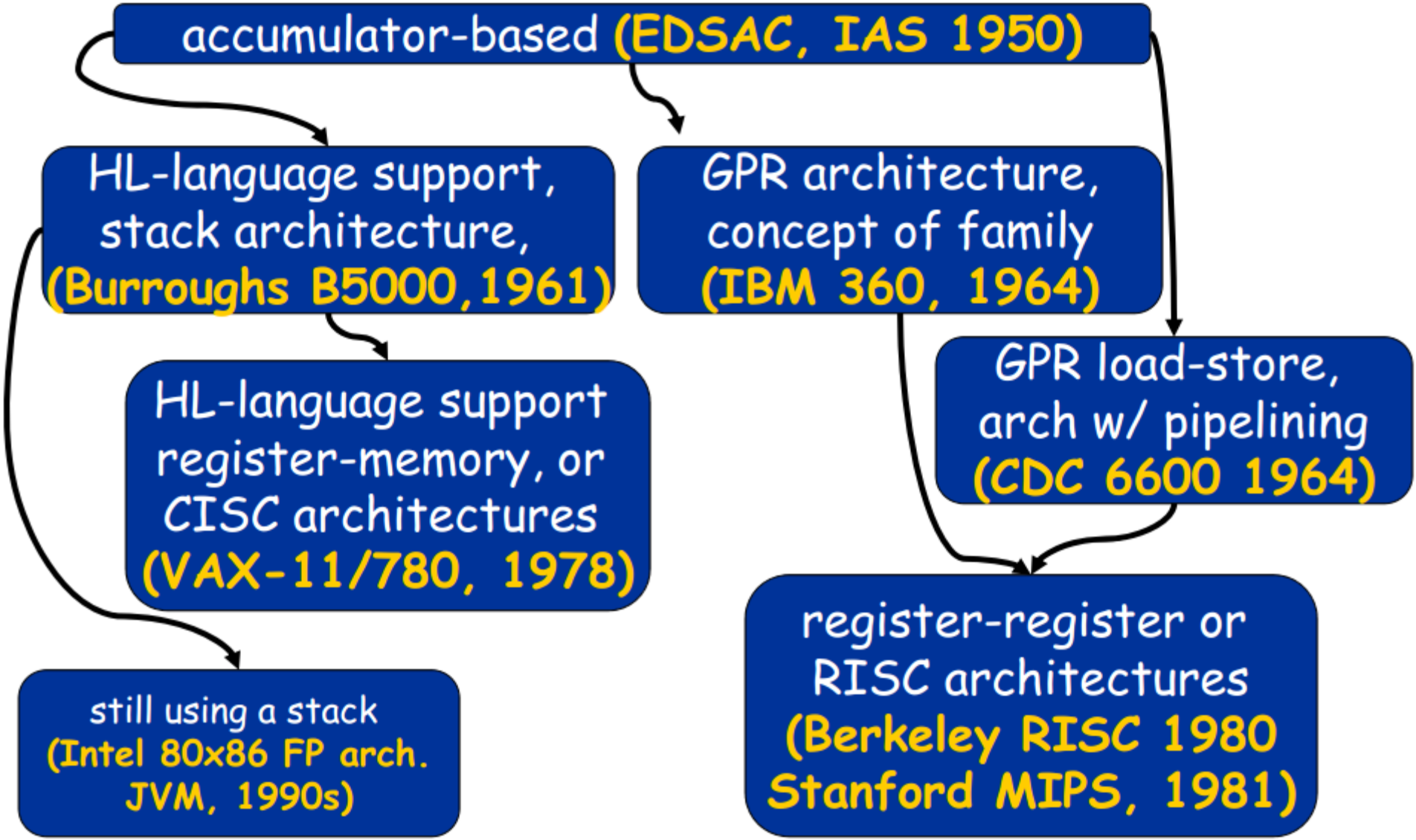
Interface Design

A good interface:

- **Lasts through many implementations (portability, compatibility)**
- **Is used in many different ways (generality)**
- **Provides *convenient* functionality to higher levels**
- **Permits an *efficient* implementation at lower levels**



Evolution of ISAs



Evolution of Instruction Sets

- Major advances in computer architecture are typically associated with landmark instruction set designs
  - Ex: Stack(B1700) vs GPR (System S/360)
- Design decisions must take into account:
  - technology(component)
  - machine organization
  - programming languages
  - compiler technology
  - operating systems
- And they in turn influence these

Did RISC win?

- RISC architectures have been successfully applied in all three main computing domains
  - desktop, server, embedded
- But one of the most, if not the most, successful architecture has been the Intel 80x86, which is not a RISC, thanks to
  - key commercial role played by binary compatibility
  - "support" offered by Moore's Law
    - (every 18 months, a smaller, faster processor…)
  - high chip volumes in the PC industry justify the big investments in such complex architectures
    - and eventually Intel Processor started using hardware to translate from 80x86 instructions to RISC-like instructions (executed internally)

ISA Design

- Design Criteria
  1. memory addressing modes
  2. operand types and sizes
  3. instruction types
  4. instructions for control-flow
  5. encoding of the instruction set
- Design Process
  1. by means of extensive simulation with a rich real benchmark suite, identify key common operations and kernels
  2. find a compromise between supporting such common cases and designing an ISA that can be implemented and validated efficiently

Number of Explicit Operands

Maximum number of operands to be specified is 3
- 2 source operands and 1 result operand

To optimize the memory bandwidth required by instructions(for fetching from Memory), the number of explicitly specified operands in the instruction needs to be reduced

- 2 operands(GPR machine)

  2 source operands(1 of the source operands is destroyed after execution to store the result)

- 1 operand(AC machine)

  1 of the operands is implied to a specific hardware register called Accumulator(AC)(result of the execution is also stored in this register)

- 0 operand(Stack machine)

  Both of the operands and the result are implied to a stack

Operand Storage

Storage

- Memory
  - - Long memory addressing
  - - Need to represent the address with a few bits
  - »Relative addressing with displacement
  - »Page/Segment addressing
- Register
  - - General purpose register
  - »Short register addressing
  - - AC
- Stack(register)
  - - Does not need for addresses

Effective Address

- Address and Physical Storage Location are two different concepts.
- Addresses of Operands are represented or implied in the instruction.
- Operand's address needs to be mapped into an Effective Address of the physical storage location

Basic Addressing Modes(A or R in instructions)

Mode	Algorithm	Advantage	Disadvantage
Immediate	opd=A	# of M refer	limited value
Direct	EA=A	simple	limited addr space
Indirect	EA=M[A]	large addr space	multiple M refer
Register	EA=R	no M refer	limited addr space
R Indirect	EA= M[R]	large addr space	extra M refer
Displacement	EA= A+[R]	flexibility	complexity
Stack	opd=S[TOP]	no M refer	limited applications

Operand Types and Sizes

- Operand type is interpreted based on opcode
- Operand type is driven by the application and usually gives implicitly its size
  - text processing
    - character: 8 bits (ASCII), 16 bits (UNICODE)
  - scientific computing (IEEE Standard 754-1985)
    - single-precision floating-point number (1 word of 32 bits)
    - double-precision floating-point number (2 words)
  - signal processing
    - 16 bit fixed-point (“low-cost floating point”: exponent is not part of the word but stored in a special variable and the DSP programmer must take care of shifting and aligning)
- Integers are represented as two’s complement binary number
  - makes signed addition easy

Instruction Types (examples for MIPS-like machines)

Type	Examples(MIPS)	
arithmetic &logical	**DADD r3, r1, r2** **DSLL r3, r1, #5**	R[r3] ←R[r1]+R[r2] R[r3] ←R[r1]<<5
floating point	**ADD.D f3, f1, f2** **DIV.D f5, f6, f7**	F[f3] ←F[f1]+F[f2] F[f5] ←F[f6]/F[f7]
data transfer	**LD r3,30(r1)** **SW r2,500(r4)** **L.Df3,100(f1)**	R[r3] $\leftarrow_{64}$ M[30+R[r1]] M[500+R[r4]] $\leftarrow_{32}$R[r2] F[f3] $\leftarrow_{64}$ M[100+F[f1]]
control	**Jr r3** **Beq R1,R2,25** **Jal 2500**	PC←R[r3] if(R[r1]==R[r2])PC←PC+4+100 RA←PC+8 PC←$PC_{64..28}$ ##10000
system	**trap**	Transfer to operating system

Measure: Top 10 Instructions for the Intel 80x86

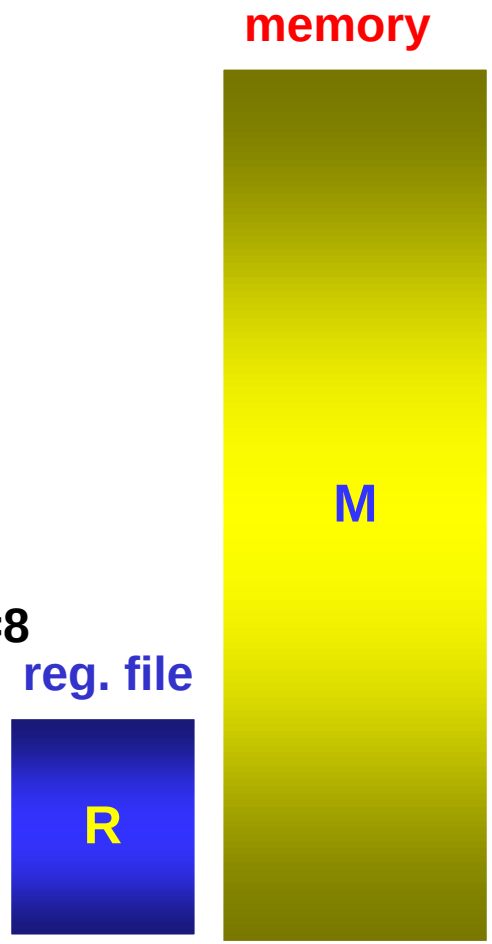
(average perc. of the five SPECint92 programs)

1	**Load**	22%
2	Conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move (reg-reg)	4%
9	call	1%
10	return	1%

these are mostly simple instructions and are responsible for 96% of all instructions executed!

Kinds of Addressing Modes

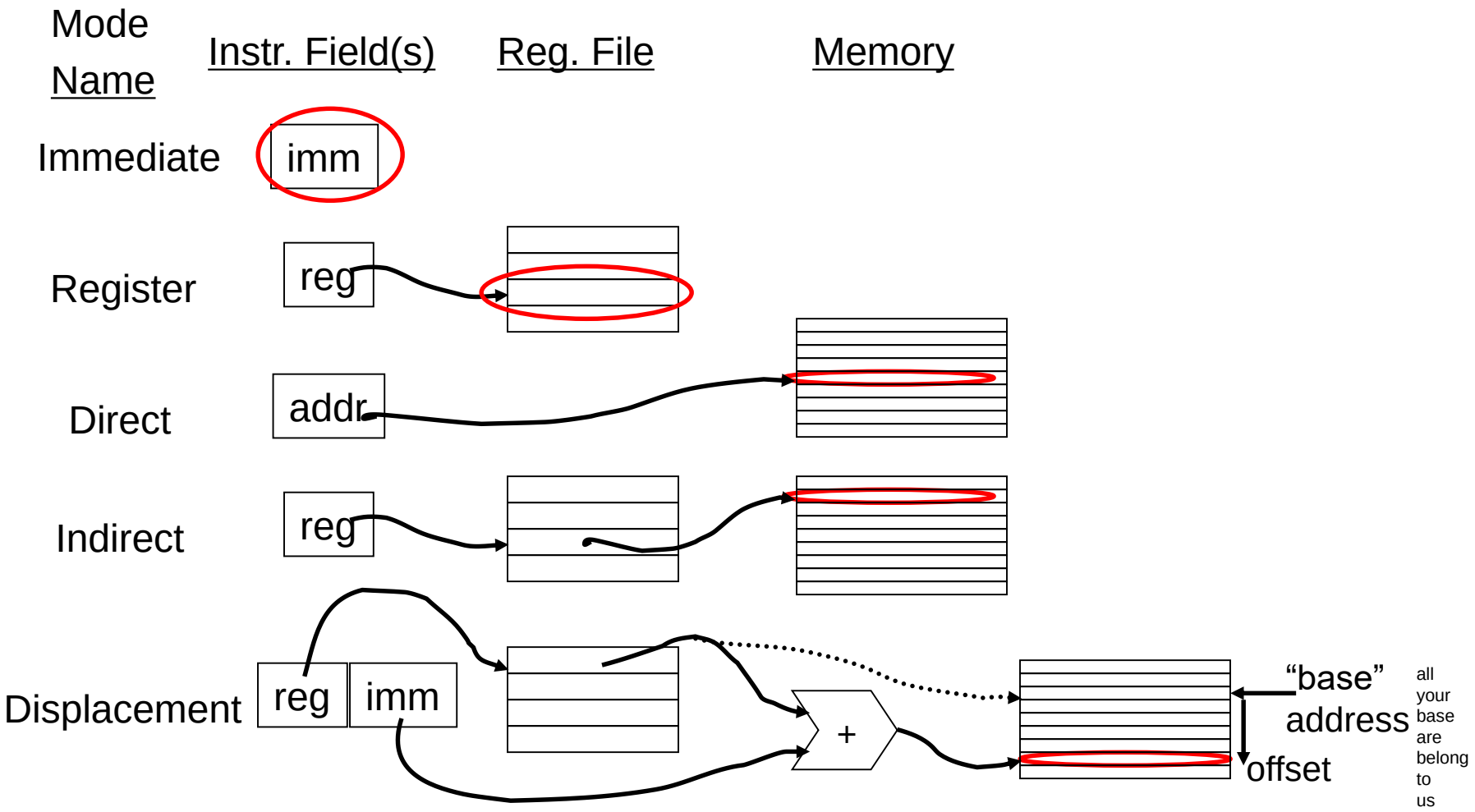
Addressing Mode	value in [ ] is the operand
• Register direct	[Ri]
• Immediate (literal)	v
• Direct (absolute)	M[v]
• Register indirect	M[[Ri]]
• Base+Displacement	M[[Ri] + v]
• Base+Index	M[[Ri] + [Rj]]
• Scaled Index	M[[Ri] + [Rj]*d + v], eg. d=8
• Autoincrement	M[[Ri]+1]
• Autodecrement	M[[Ri] - 1]
• Memory Indirect	M[M[Ri]]
• [Indirection Chains]	



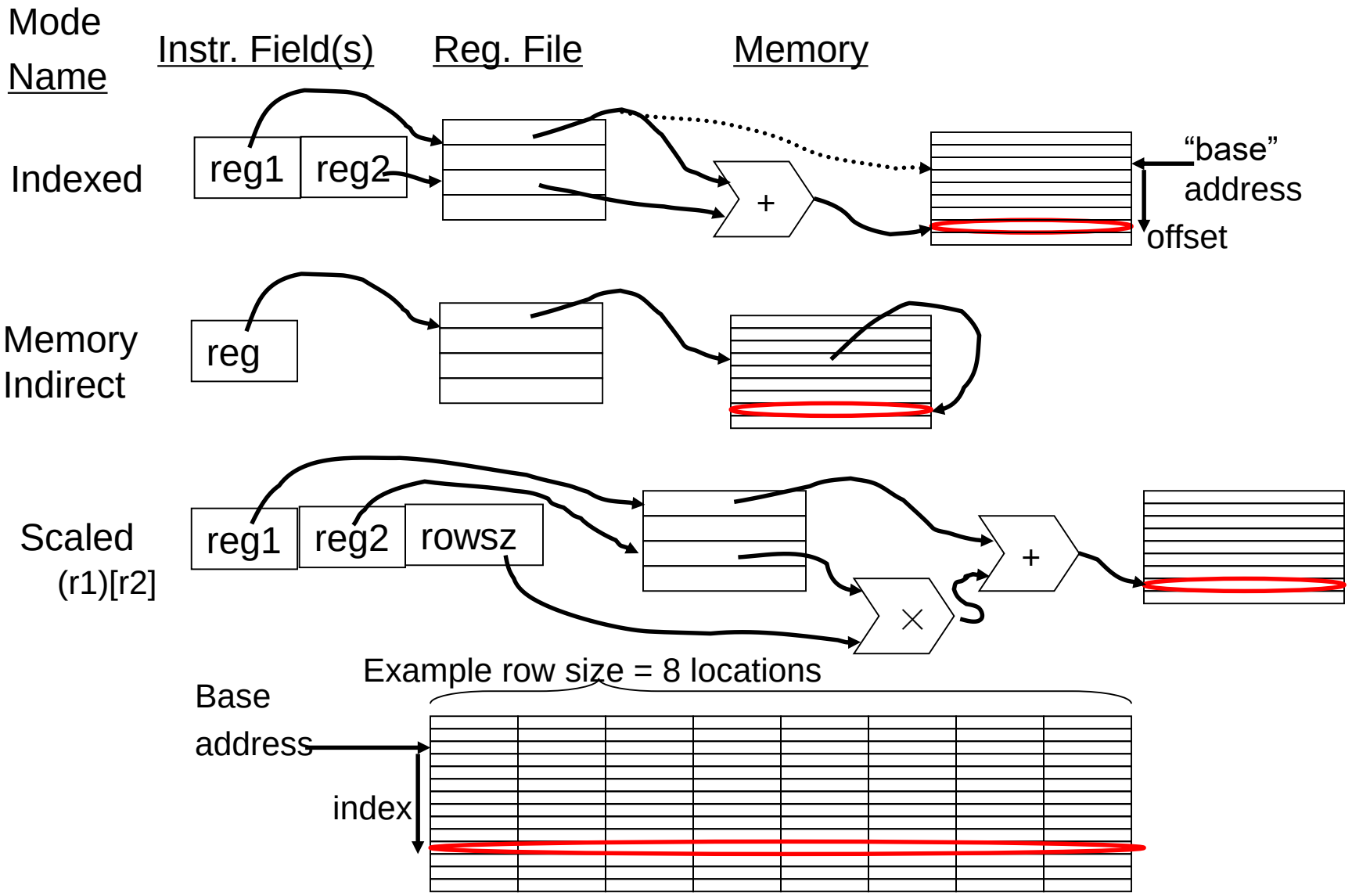
Memory Addressing Modes

mode1	example	meaning
register	**Add r3,r1**	R[r3]←R[r3]+R[r1]
immediate	Add r3,#7	R[r3] ←R[r3]+7
displacement	**Add r3,100(r1)**	R[r3] ←R[r3]+M[100+R[r1]]
reg.indirect	**Add r3,(r1)**	R[r3] ←R[r3]+M[R[r1]]
indexed	**Add r3,(r1+r2)**	R[r3] ←R[r3]+M[R[r1]+R[r2]]
direct/absolute	**Add r3,(#1001)**	R[r3] ←R[r3]+M[1001]
mem.indirect	**Add r3,@(r1)**	R[r3] ←R[r3]+M[M[R[r1]]]
autoincrement	**Add r3,(r2)++**	R[r3] ←R[r3]+M[R[r2]] R[r2] ←R[r2]+d
autodecrement	**Add r3,--(r2)**	R[r2] ←R[r2]-d R[r3] ←R[r3]+M[R[r2]]
scaled	**Add r3,100(r1)[r2]**	R[r3] ←R[r3]+M[100+R[r2]+R[r1]*d]

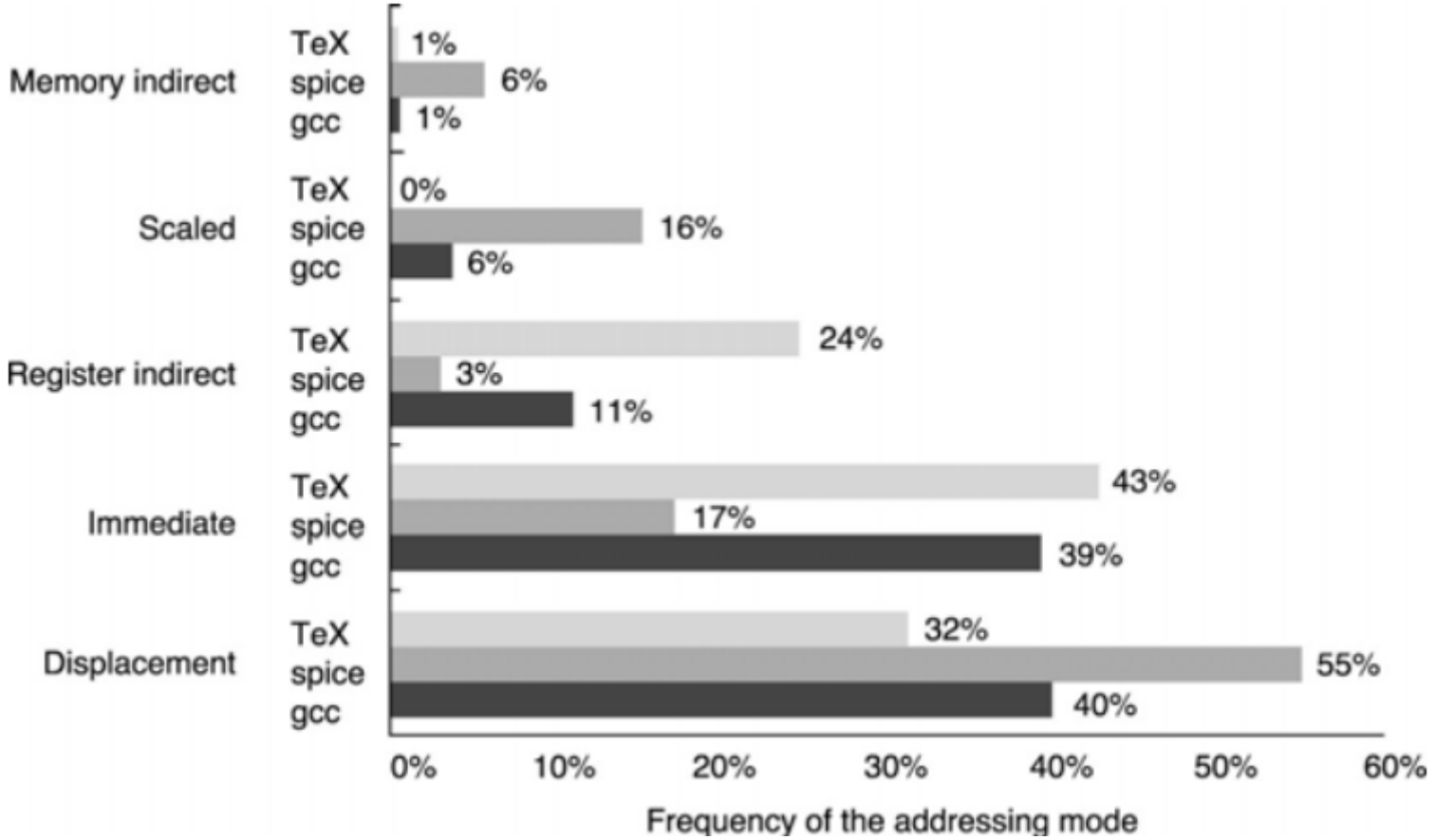
Addressing Modes Visualization



Addressing Modes Visualization Cont.



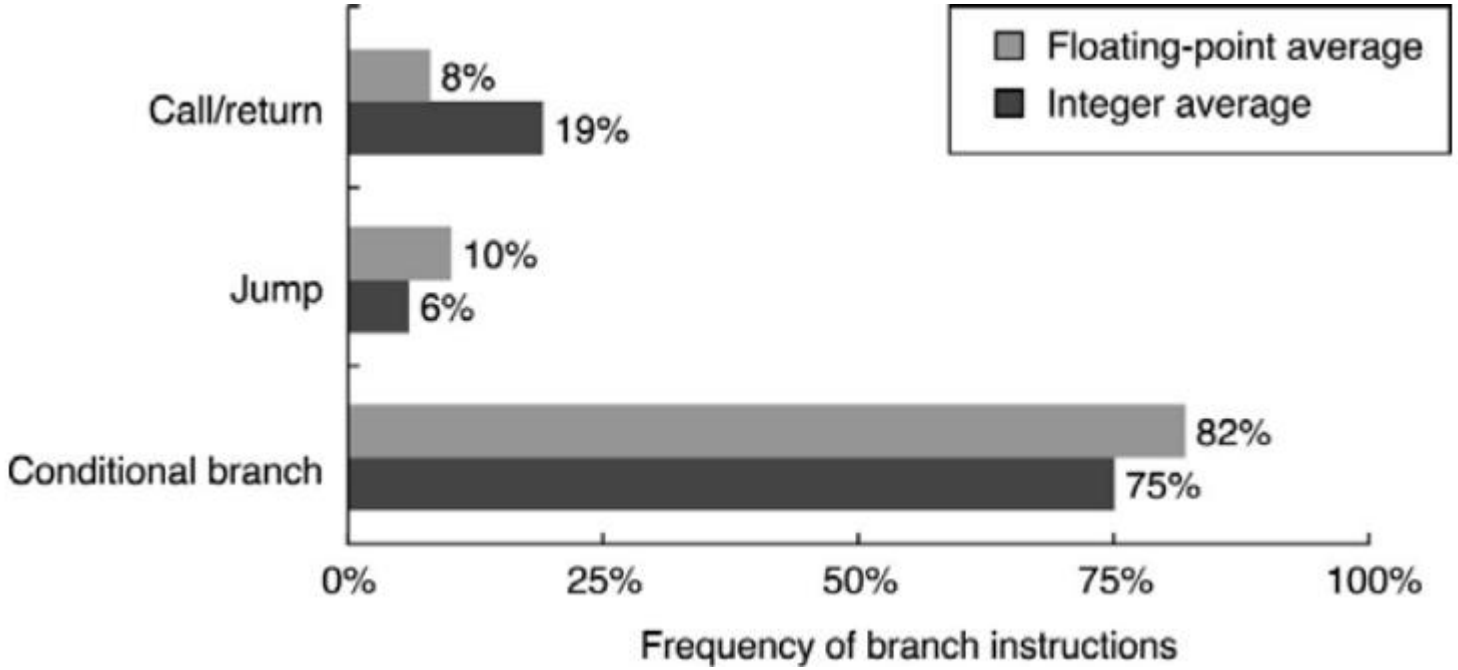
Memory Addressing Mode - Use



- Figure A.7: measuring usage patterns on the VAX
  - these five modes account for 97-99% of all memory accesses
  - memory addressing account for 50% of all operand references

Measure: Instruction for Control Flow

(breakdown for Alpha μP with SPEC CPU2000)



- Figure A.14: conditional branch dominates. Nowadays three techniques are mostly used to implement them
  - condition-code (test special bits set by ALU operations)
  - condition register (test arbitrary register with result of a comparison)
  - compare & branch (compare operation is part of the branch instruction

Encoding an Instruction Set: Variable vs. Fixed

- variable encoding
  - separate address specifier determines addressing modes for that operand
- fixed encoding
  - addressing mode is within opcode: one memory operand and a couple of addressing modes
- trade-off
  - variable encoding gives better code size
    minimize bit numbers to represent a program (instruction lengths vary widely)
  - fixed encoding gives better performance
    easier decoding, faster pipeline