Computer Architecture (Fall 2022)

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.

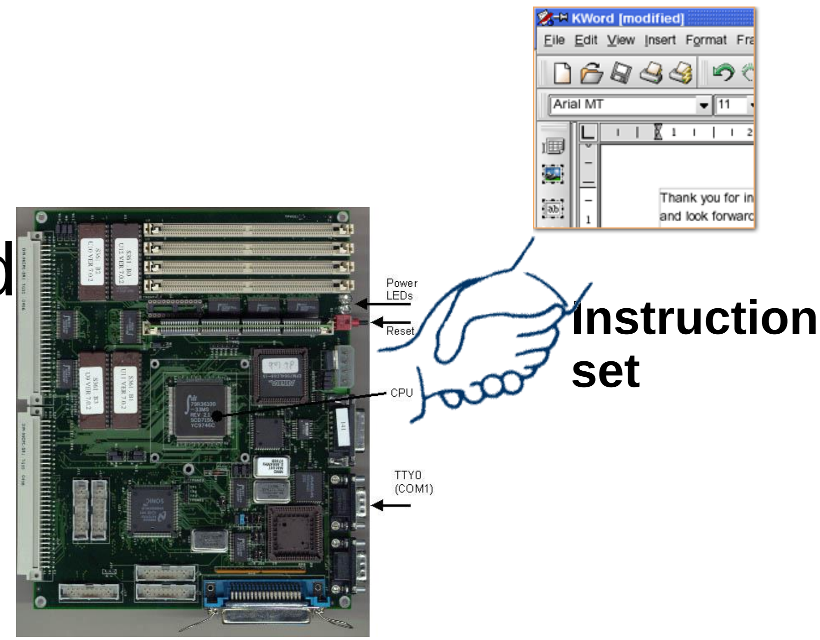
- Early 1960s: IBM launches the IBM 360 , a landmark ISA
  - IBM coins the term computer architecture to refer to the program-visible portion of the instruction set
    - today the term has a broader meaning (design the whole computer system)
  - a family of computers that are able to run the same software
    - 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

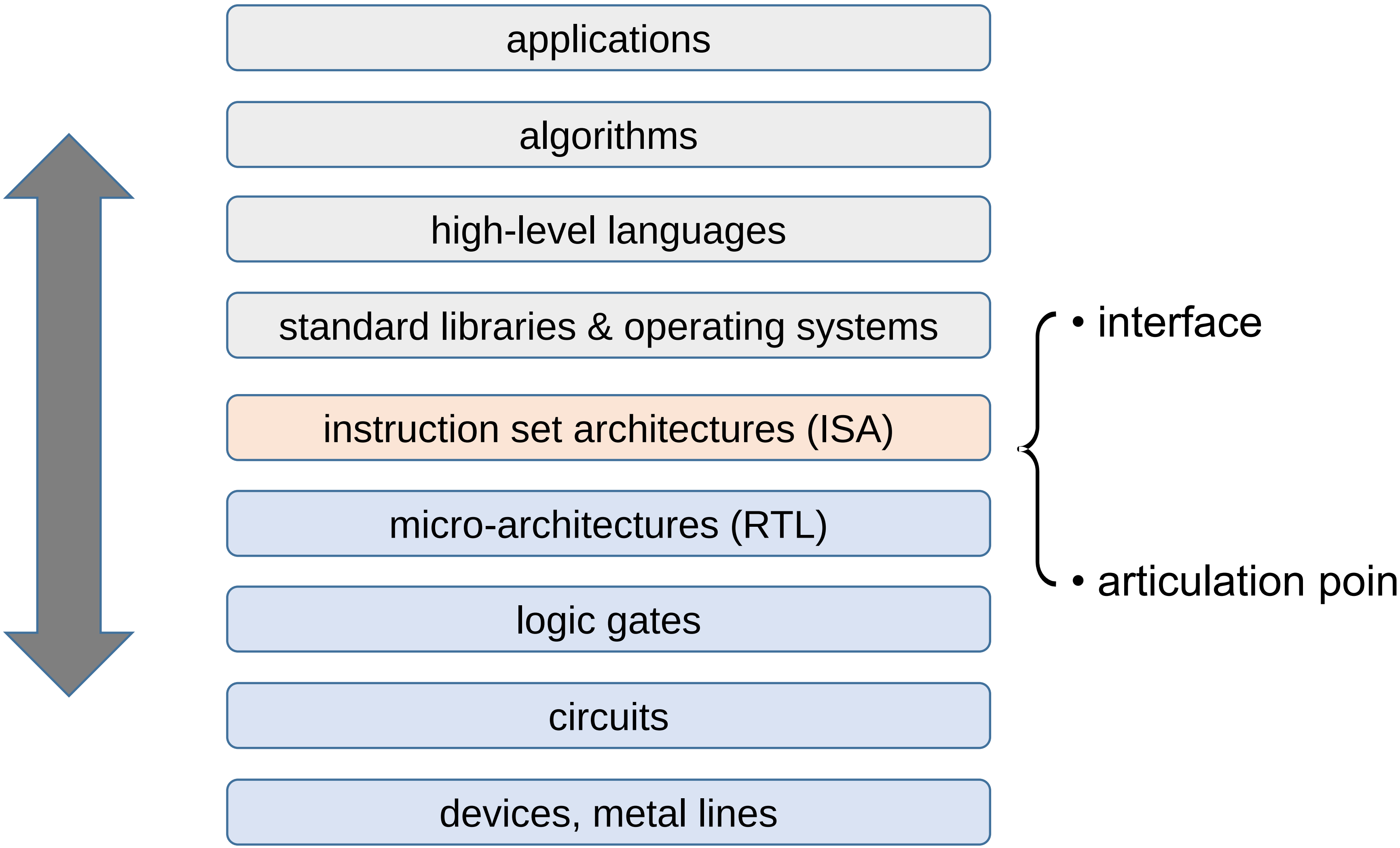
- Desktop Computing – emphasizes performance of programs with integer and floating point data types; little regard for program size or processor power
- Servers - used primarily for database, file server, and web applications; FP performance is much less important for performance than integers and strings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

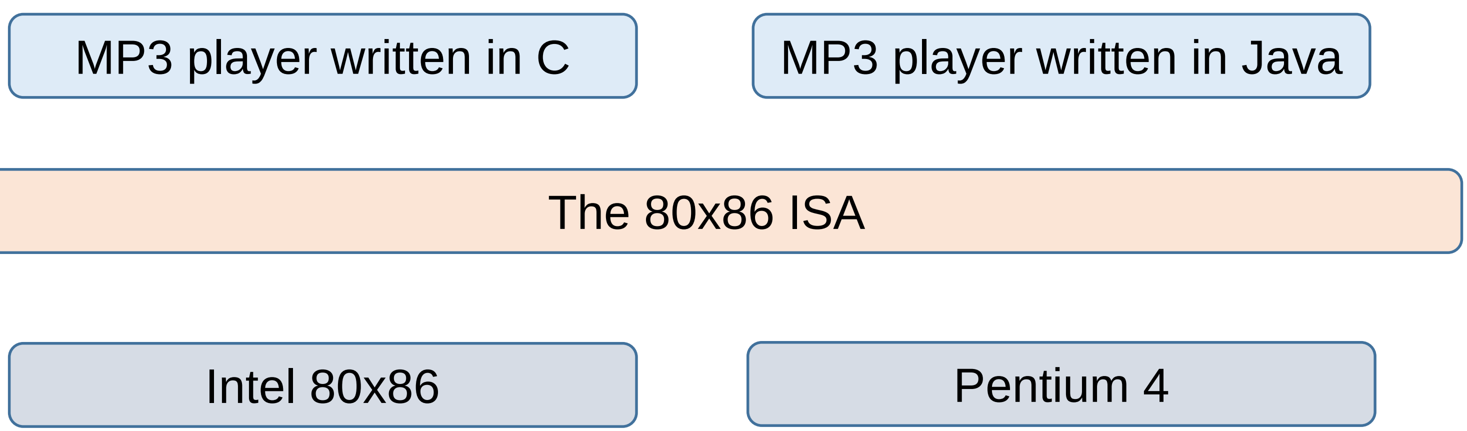


Computer Architecture and Abstraction Layers



Computer Architecture and Abstraction Layers

- 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



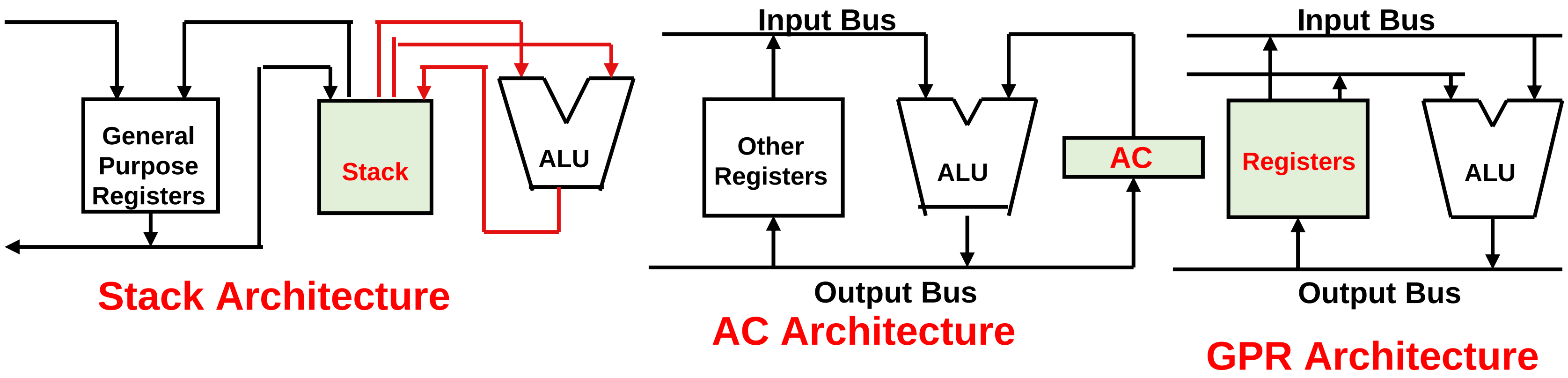
ISA: Interface between HW & SW

- 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

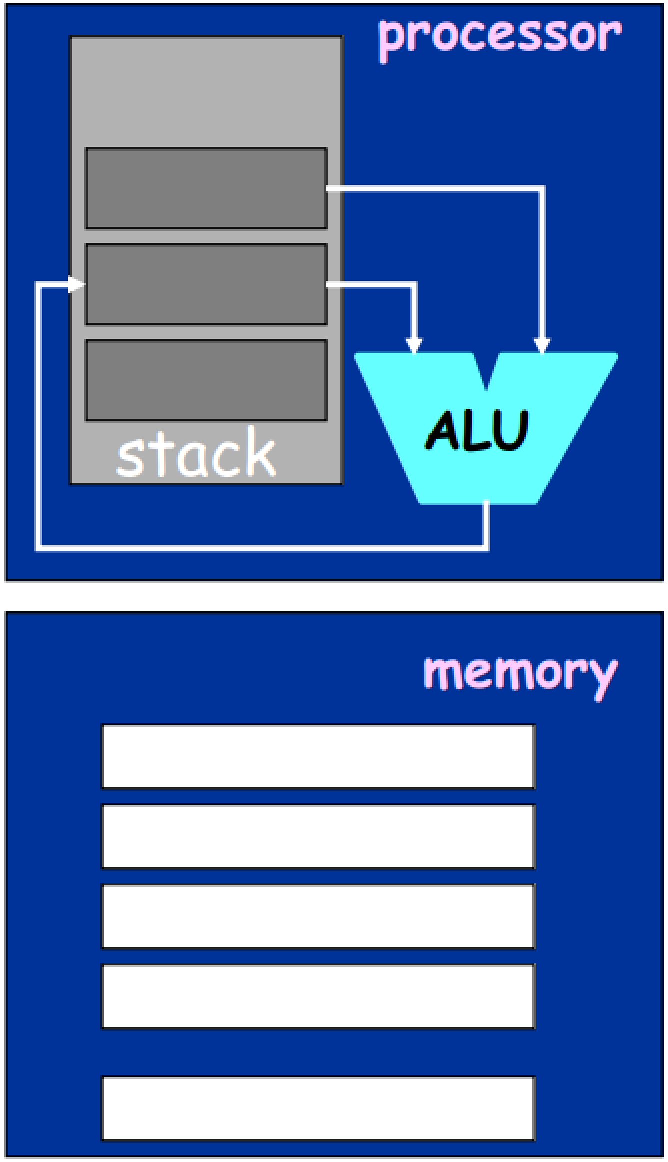


Taxonomy of ISAs: Stack

a = b + (c * d)

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

- Instruction operands
  - – none (implicit) for ALU operations
  - - one for transfer from/to memory
    - push/pop
- Pros
  - – short instruction
  - – simple compiler
- Cons
  - – inefficient code
    - many swaps, copies
  - – stack may be slow
- Examples
  - B5000, JVM



Taxonomy of ISAs: Stack

- Instruction set:

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

  push A, pop A

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

push a

push b

*

push a

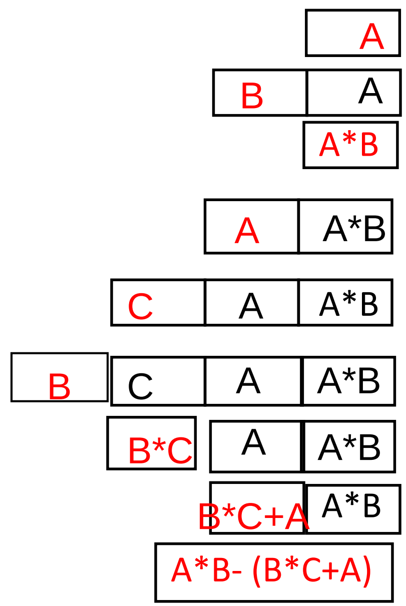
push c

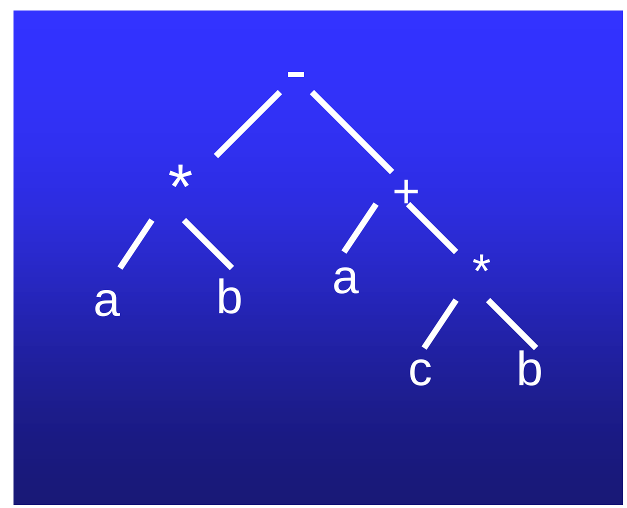
push b

*

+

-



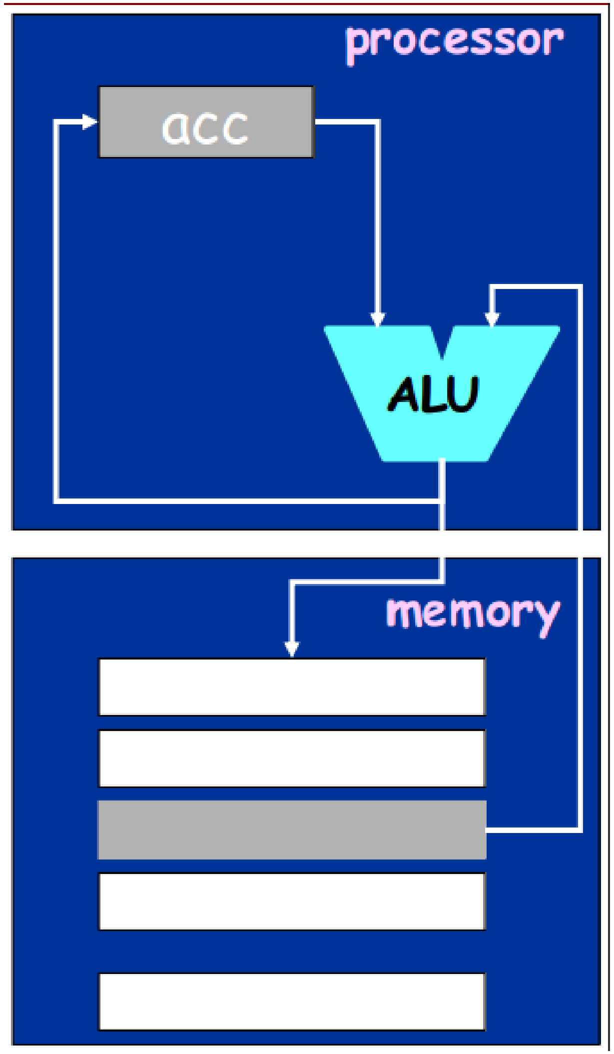


Taxonomy of ISAs: Accumulator

a = b + (c * d)

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

- 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)

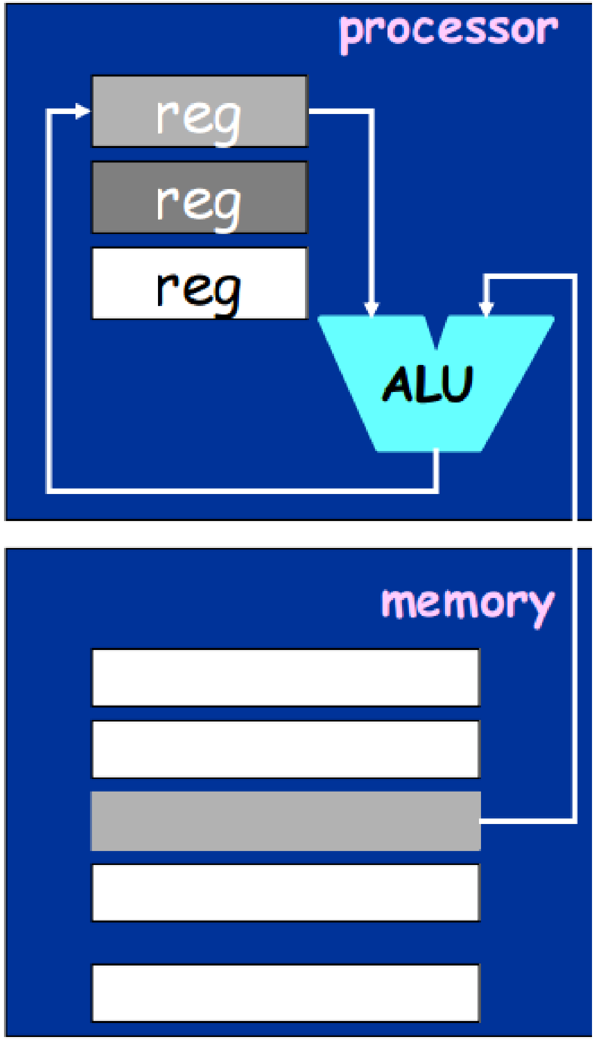


Taxonomy of ISAs: Register-Memory

a = b + (c * d)

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

- 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

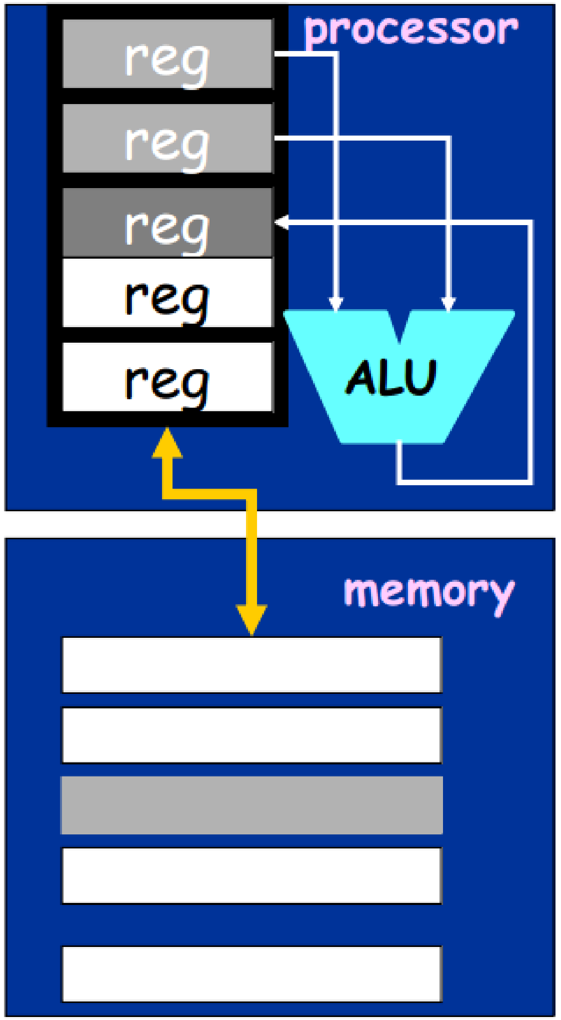


Taxonomy of ISAs: Register-Register (or Load-Store)

a = b + (c * d)

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

- 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

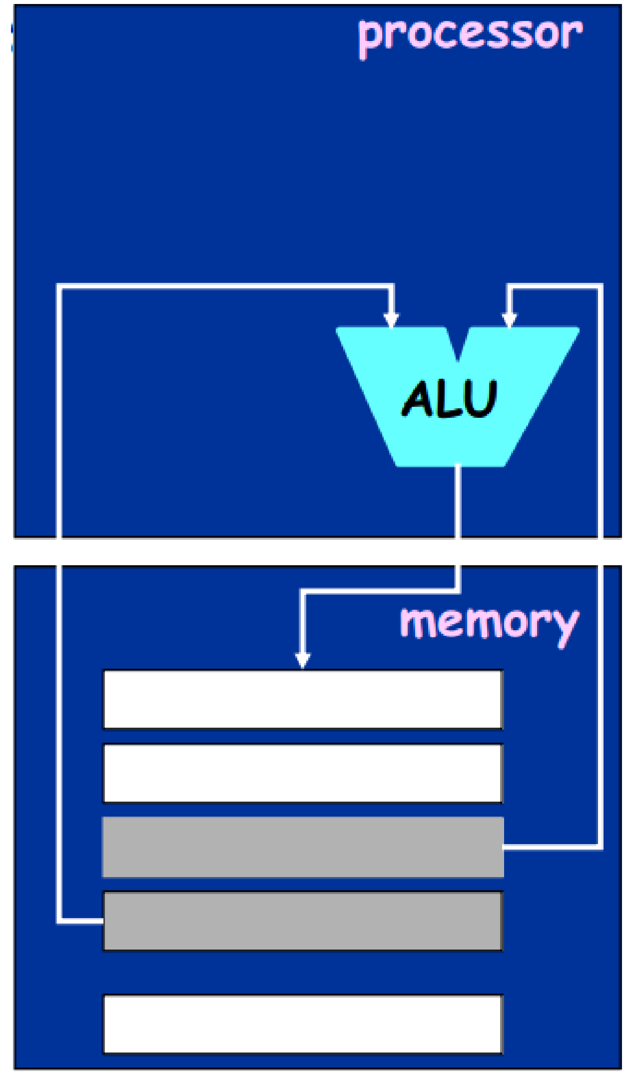


Taxonomy of ISAs: Memory-Memory

a = b + (c * d)

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

- 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



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	$R[R1] \leftarrow R[R2] + R[R3]$
(1,2):	ADD	R1, X	$R[R1] \leftarrow R[R1] + M[X]$
(3,3):	ADD	X1,X2,X3	$M[X1] \leftarrow M[X2] + M[X3]$

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	Maximim No of operands allowed	Examples
(0, 3)	0	3	SPARC, MIPS, PowerPC, ALPHA
(1, 2)	1	2	Intel 80x86, Mototola 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

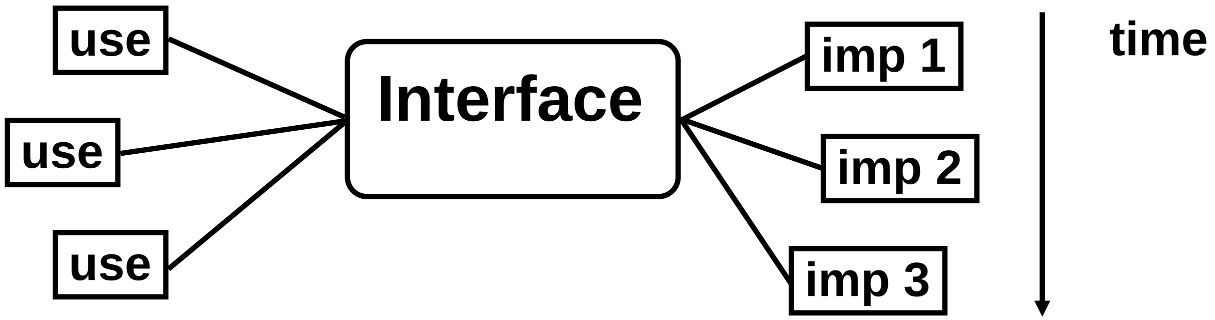
```
LD      R1,A
ADD     R1,B
ADD     R1,C
```

RM instructions reduce IC

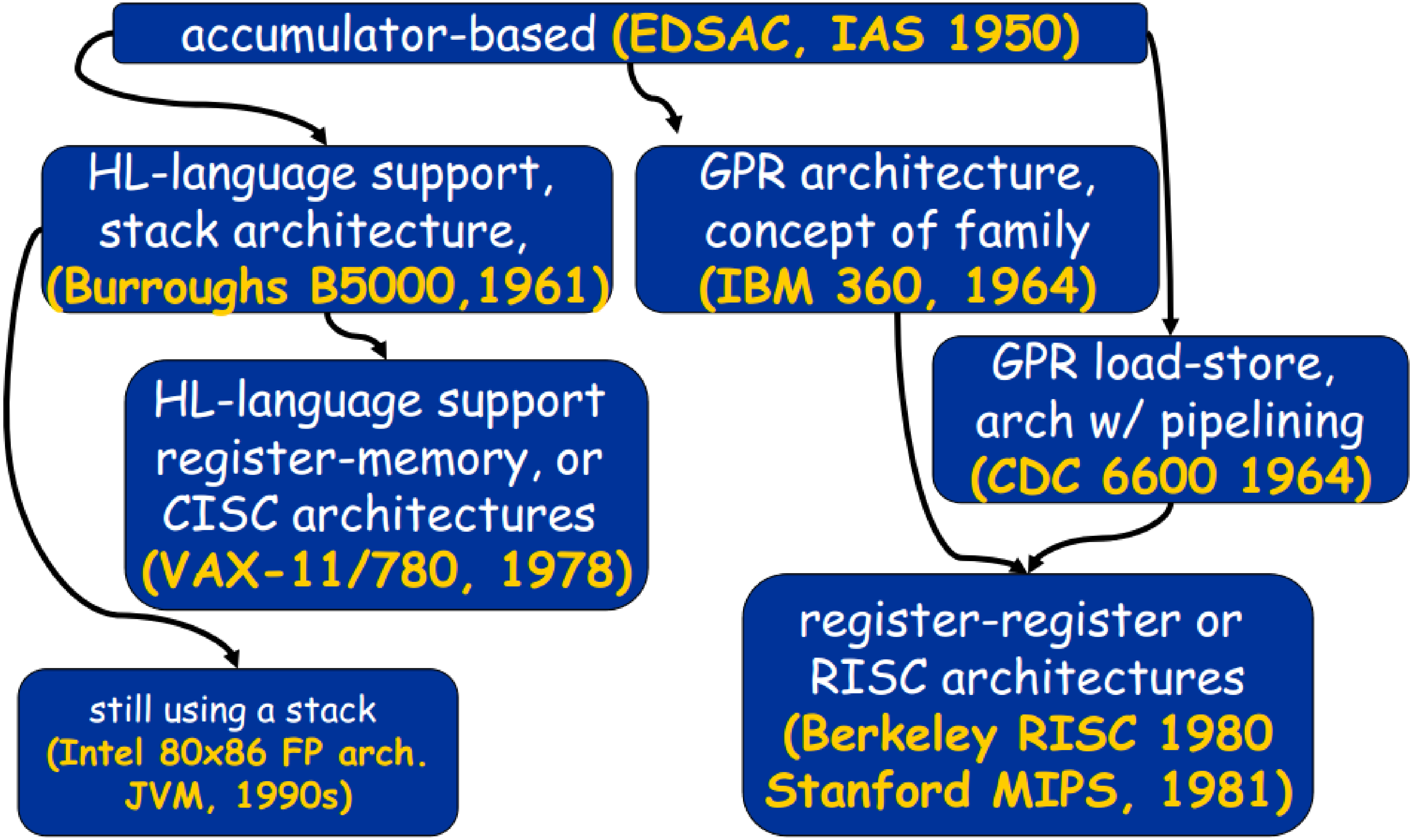
Interface Design

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**	$R[r3] \leftarrow R[r1]+R[r2]$ $R[r3] \leftarrow R[r1]<<5$
floating point	**ADD.D f3, f1, f2** **DIV.D f5, f6, f7**	$F[f3] \leftarrow F[f1]+F[f2]$ $F[f5] \leftarrow F[f6]/F[f7]$
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**	$PC \leftarrow R[r3]$ $if(R[r1]==R[r2])PC \leftarrow PC+4+100$ $RA \leftarrow PC+8$ $PC \leftarrow PC_{64..28}$ ##10000
system	**trap**	Transfer to operating system

Measure: Top 10 Instructions for the Intel 80x8 (average perc. of the five SPECint92 programs)

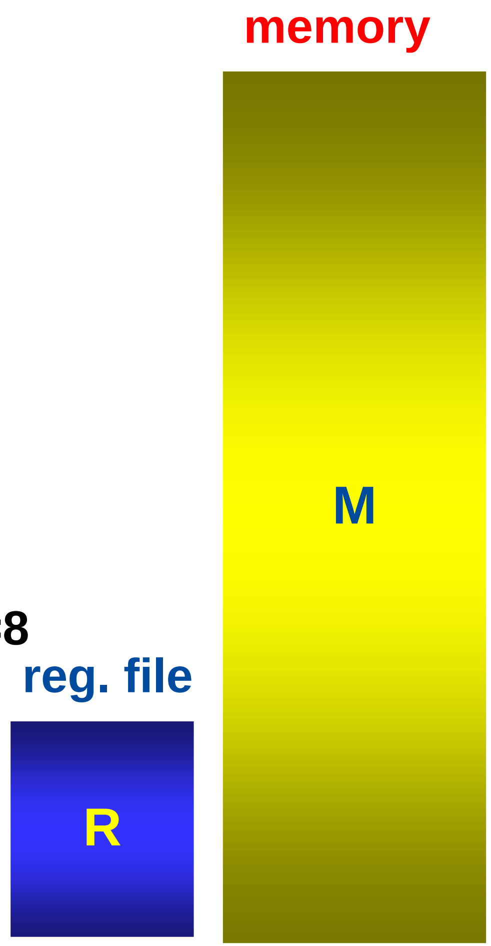
1	**Load**	22%
2	Conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move (reg-reg)	4%
9	call	1%
10	return	1%

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

Kinds of Addressing Modes

OP	Ri	Rj	v

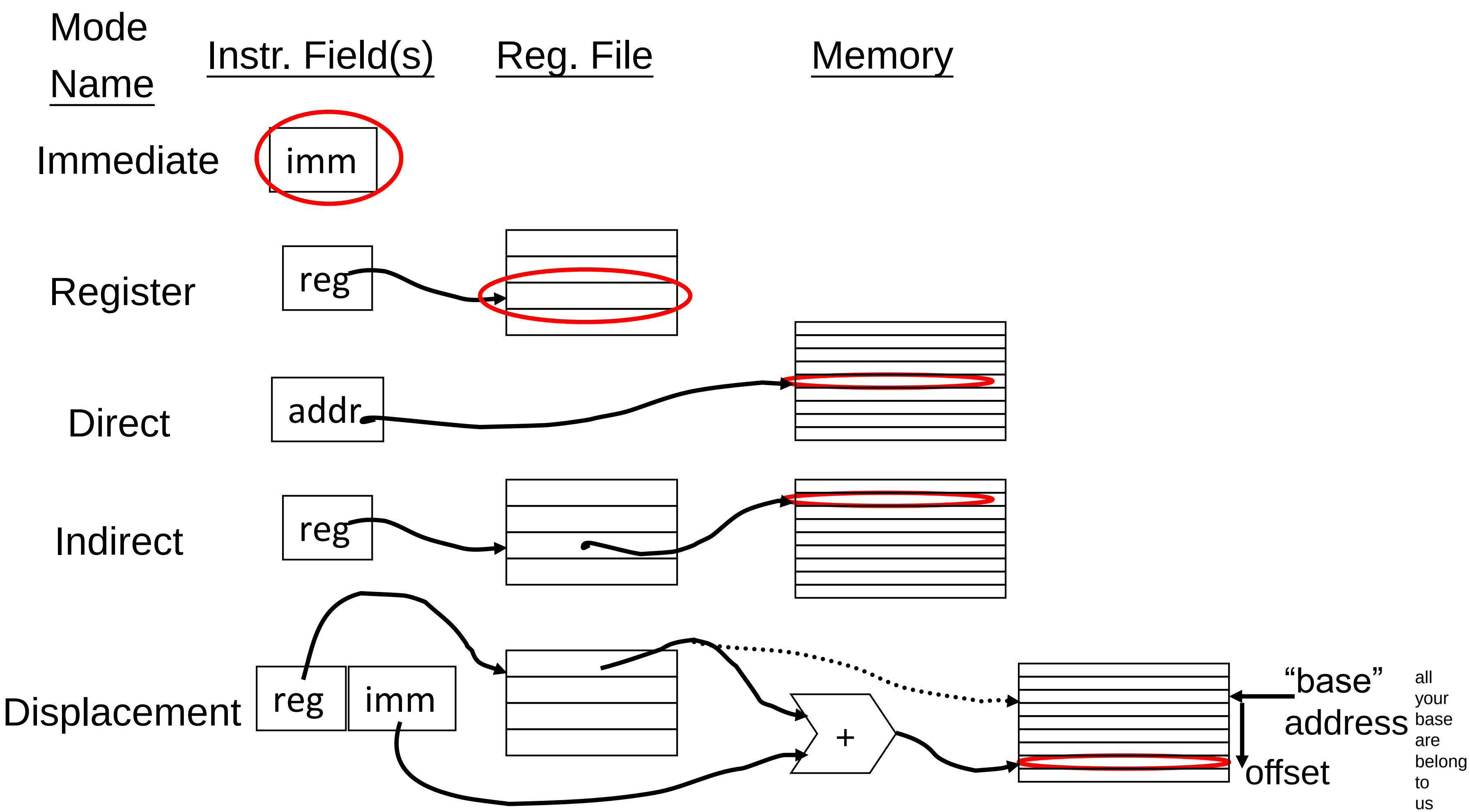
Addressing Mode	value in [ ] is the operand
• Register direct	[Ri]
• Immediate (literal)	v
• Direct (absolute)	M[v]
• Register indirect	M[[Ri]]
• Base+Displacement	M[[Ri] + v]
• Base+Index	M[[Ri] + [Rj]]
• Scaled Index	M[[Ri] + [Rj]*d + v], eg. d=8
• Autoincrement	M[[Ri]+1]
• Autodecrement	M[[Ri] - 1]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



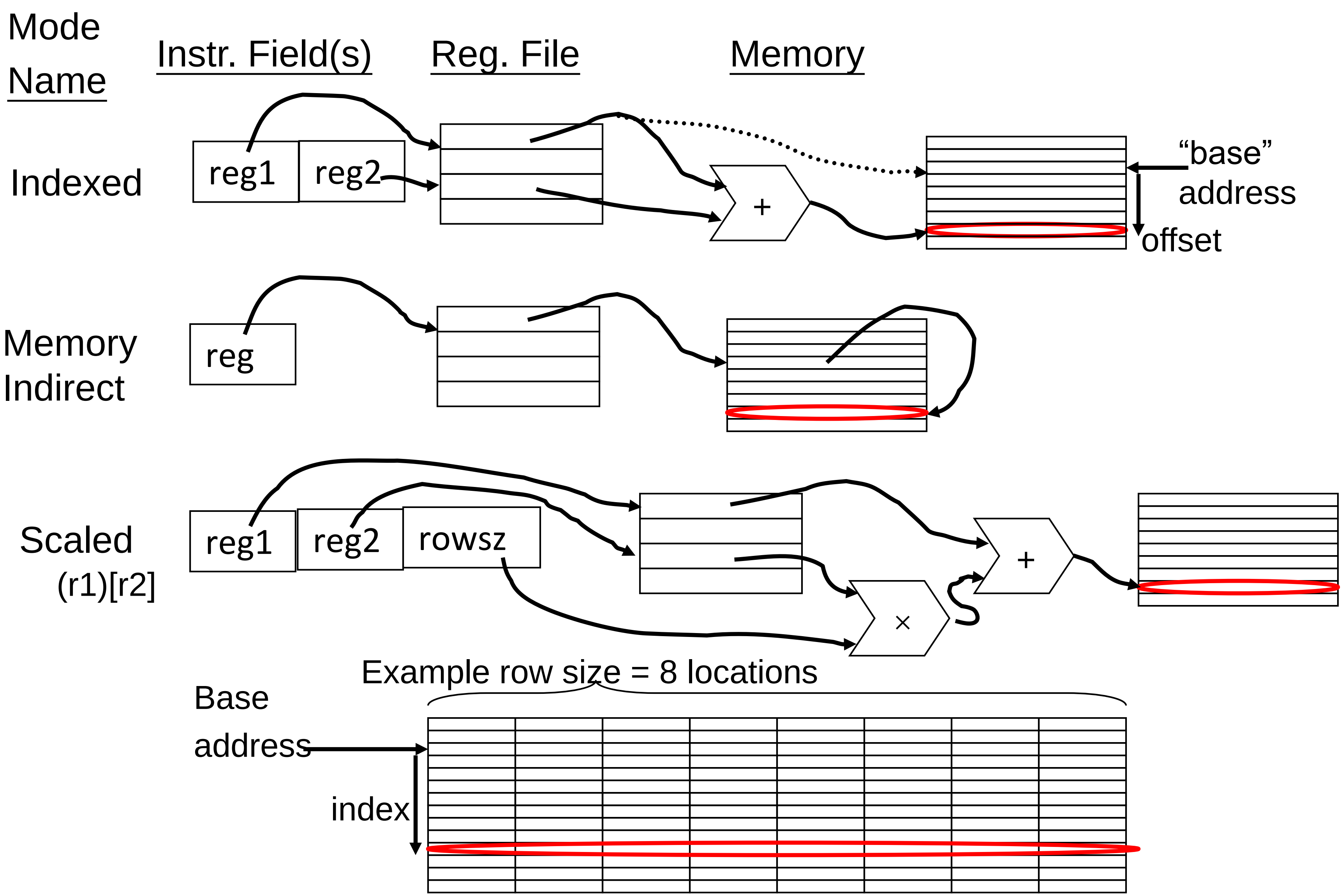
Memory Addressing Modes

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]

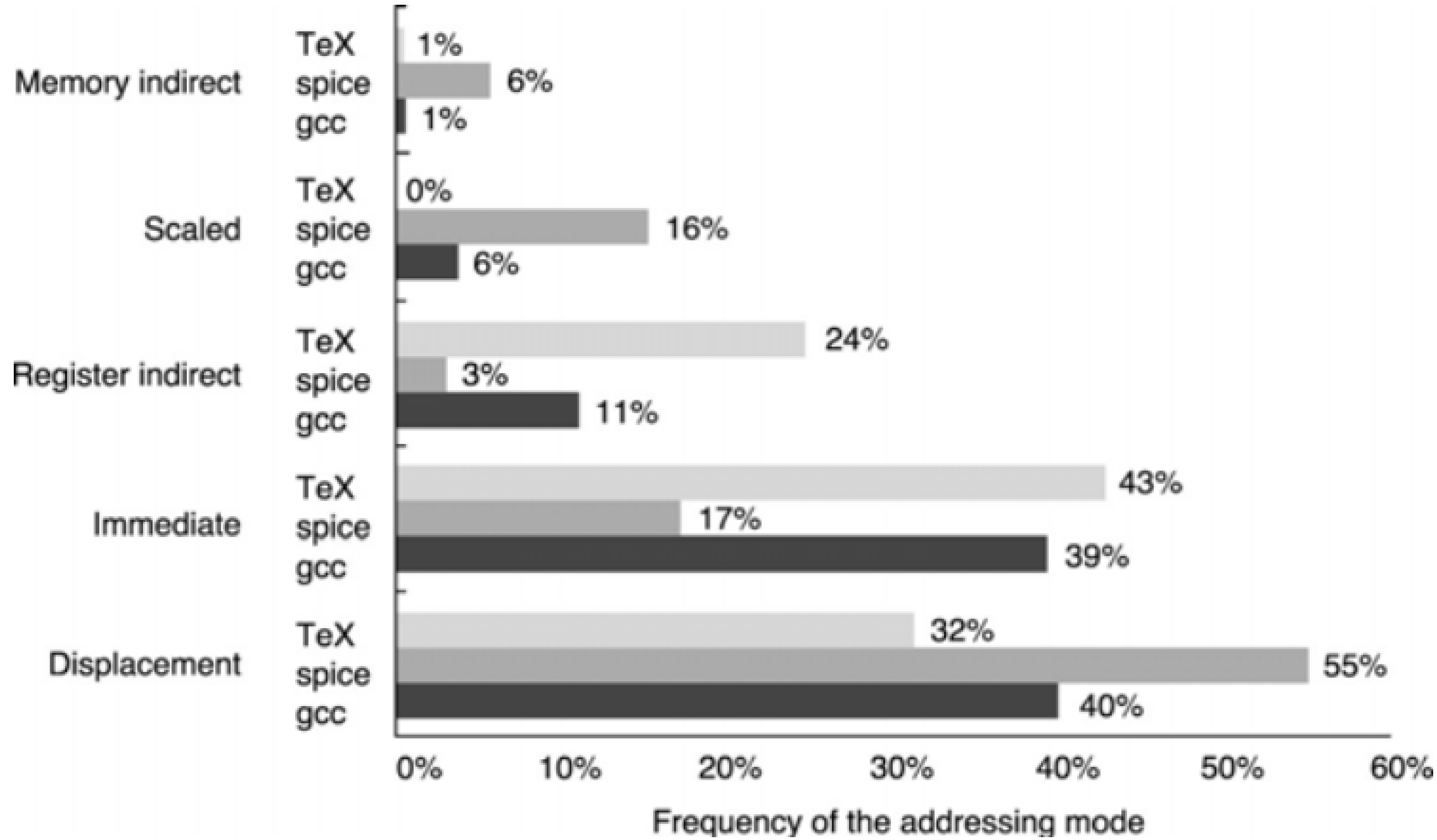
Addressing Modes Visualization



Addressing Modes Visualization Cont.

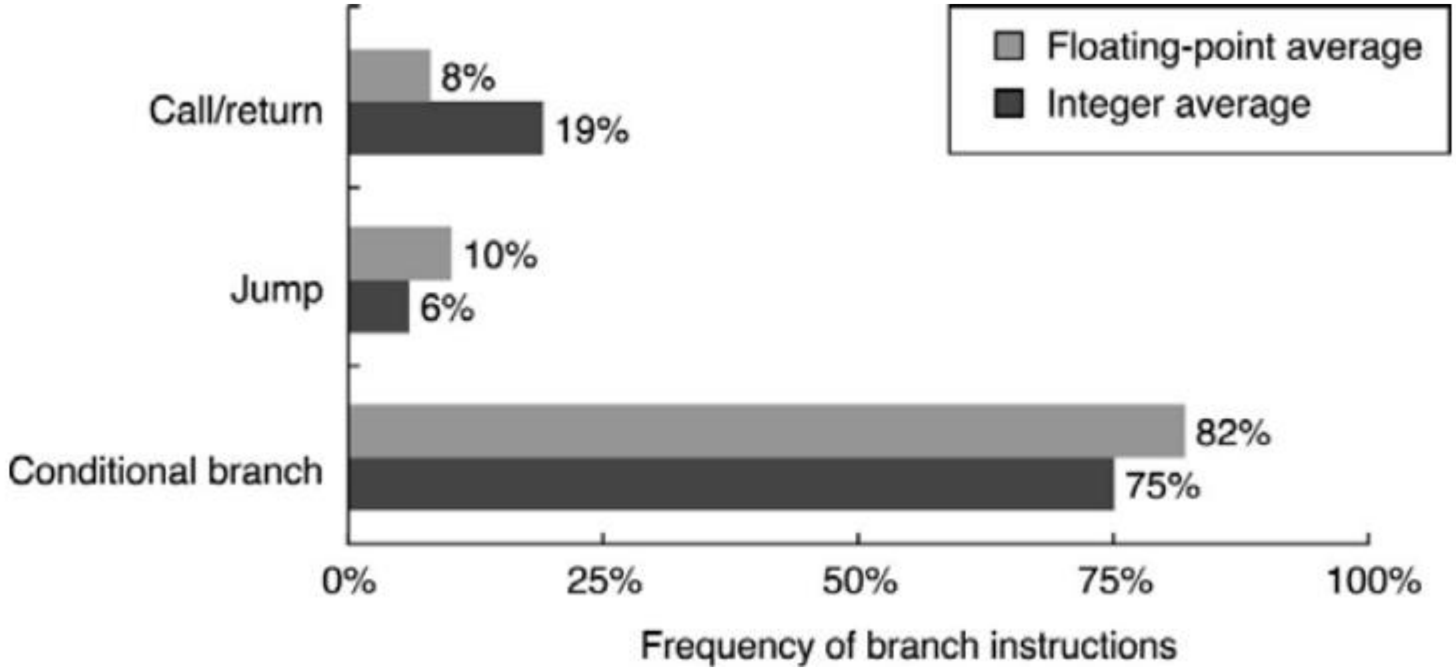


Memory Addressing Mode - Use



- 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 P 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

MIPS

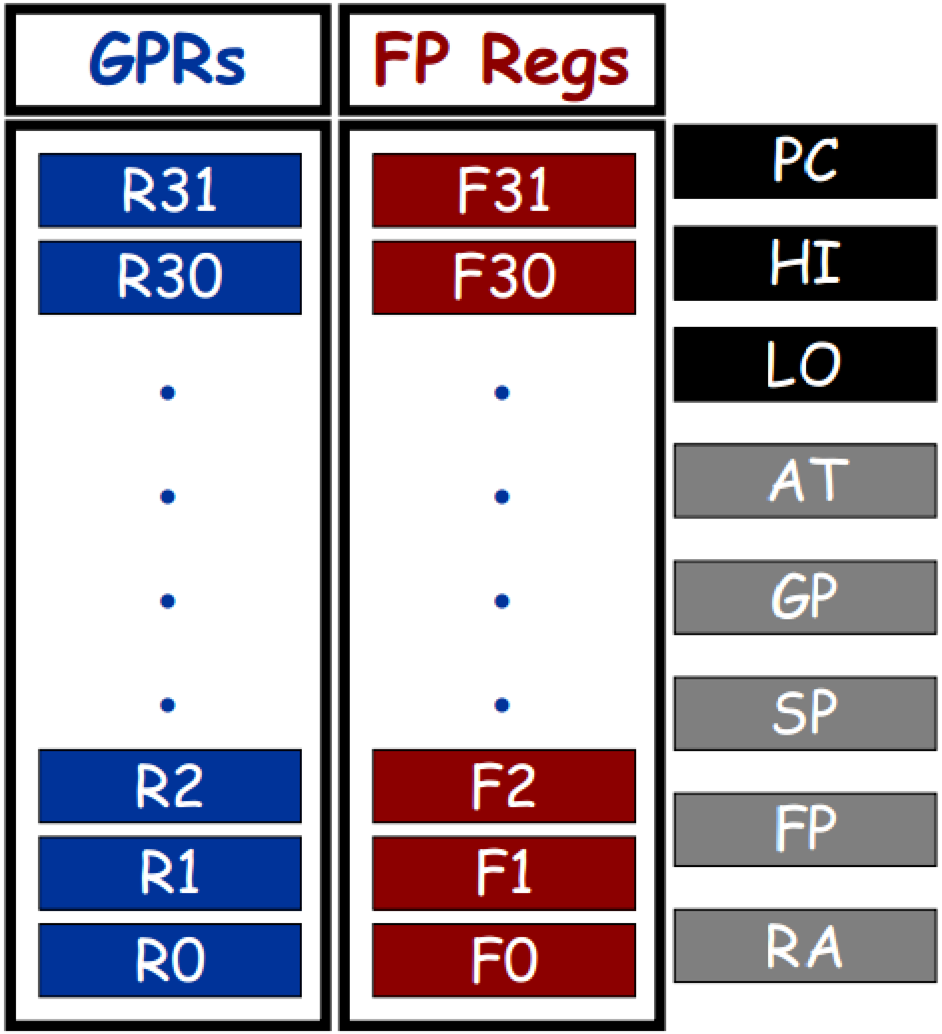
- 1981: Stanford MIPS Computer, a "Microprocessor without Interlocked Pipeline Stages" [Hennessy81]
  - no HW to stall the pipeline (handling dependencies is compiler's job)
- 1984: Hennessy founds MIPS Computer System
  - R2000 & R3000 first products (with interlock in HW!)
- 1991: MIPS releases the 64-bit R4000
  - SGI buys MIPS which becomes the division MIPS Technologies
- 1999: MIPS-Technologies so successful that SGI spins off it
  - two products MIPS32, MIPS64
  - major revenues portion from licensing the design
  - estimated 1/3 of the produced RISCs is a MIPS-based design
    - almost 100 million of MIPS manufactured in 2002
    - Used in products from ATI Technologies, Broadcom, Cisco, NEC, Nintendo SGI, Sony, Texas Instruments, Toshiba…
- MIPS64 is used throughout the textbook to illustrated the ILP techniques

Before Introducing MIPS64: What We Expect?

- Architecture?
  - register-to-register (load-store)
- Registers?
  - "many" general purpose registers and dedicated FP registers
- Addressing modes support?
  - displacement (address-offset of 12 to 16 bits)
  - using PC-relative addressing, branch +/- 2^15 <u>words</u> from PC
  - immediate (size 8-16 bits)
  - register indirect
- Data sizes & types?
  - Integers (8,16,32,64 size) and FP (64)
- Instructions types?
  - strong support for simple pervasive instructions
    - load, store, add, sub, move register-register, and shift
    - compare equal/not equal/less, branch, jump, call, return
- Encoding?
  - fixed for performance rather than variable for code size

MIPS - Registers

- 32 64-bit GPRs
  - R0 is hardwired to zero
  - AT, GP, SP, FP, RA are conventionally mapped to R1, R28, R29, R30, R31
- 32 Floating Point Registers holding either
  - 32 single-precision values (32 bits)
    - half of the FPR is not used
  - 32 double-precision values (64 bits)
- Special registers
  - PC – program counter
  - HI, LO
    - integer multiply result (higher and lower word)
    - integer divide result (remainder and quotient)



MIPS – Data Types

- Integer data
  - 8-bit bytes
  - 16-bit half words
    - present in C and popular in programs like operating systems (concerned about size of data structures)
    - also even more popular if Unicode becomes more pervasive
  - 32-bit words
  - 64-bit double words
- Floating Point
  - 32-bit single precision
    - Same motivations as for the 16-bit half words
  - 64-bit double precision

These are loaded onto the GPRs with either additional zeros or the sign bit and operated upon with 64-bit integer operations

MIPS64 operations work directly on these

MIPS – Addressing Modes

- The hardware only support two addressing modes
  - **immediate** (16-bit)

    DADDI R3, R2, #7 ;   Regs[R3] ←Regs[R2] + 7

  - **displacement** (16-bit)

    LD R3, 100(R1) ;   Regs[R3] ←Mem[100+Regs[R1]]

- Two other modes supported "indirectly"
  - **register indirect** (placing 0 in the 16-bit displacement field)

    LD R3, 0(R1) ;   Regs[R3] ←Mem[Regs[R1]]

  - **direct addressing** (using 0 as the base register)

    LD R3, 1001(R0) ;   Regs[R3] ←Mem[1001]

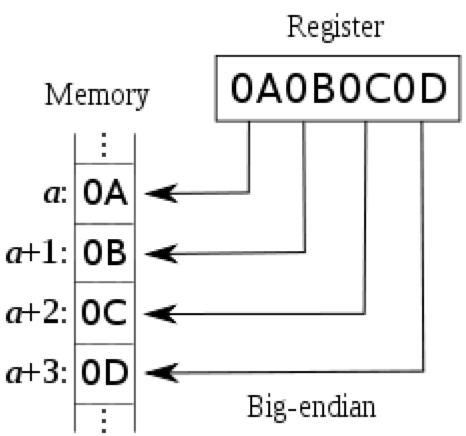
- Memory is byte-addressable with a 64-bit address

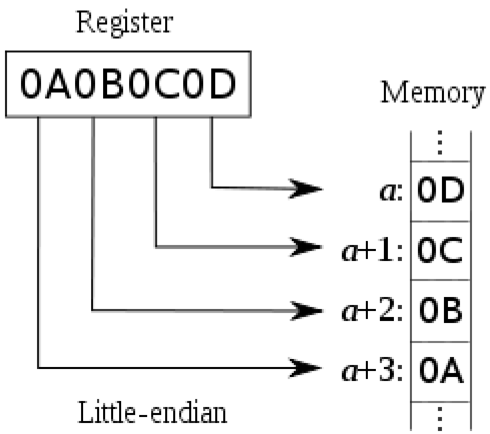
Interpreting Memory Addresses

- Memory word is addressed by the byte at the lowest address

- Big Endian
  - Most significant byte is at the lowest address
  - "big end first"
  - Ex: Motorola 68000, SPARC, IBM 360

- Little Endian
  - Least significant byte is at the lowest address
  - "little end first"
  - Ex: Intel x86, AMD64, VAX





Some architectures can be configured in both ways: ARM, MIPS, IA64, PowerPC

MIPS – Instruction Format and Layout

- MIPS instructions fall into 5 classes:
  - Arithmetic/logical/shift/comparison
  - Control instructions (branch and jump)
  - Load/store
  - Other (exception, register movement to/from GP registers, etc.)

- Three instruction encoding formats:
  - R-type (6-bit opcode, 5-bit rs, 5-bit rt, 5-bit rd, 5-bit shamt, 6-bit function code)
  - I-type (6-bit opcode, 5-bit rs, 5-bit rt, 16-bit immediate)
  - J-type (6-bit opcode, 26-bit pseudo-direct address)

Example Instructions

- ADD \$2, \$3, \$4
  - R-type A/L/S/C instruction
  - Opcode is 0's, rd=2, rs=3, rt=4, func=000010
  - 000000 00011 00100 00010 00000 000010

- JALR \$3
  - R-type jump instruction
  - Opcode is 0's, rs=3, rt=0, rd=31 (by default), func=001001
  - 000000 00011 00000 11111 00000 001001

- ADDI \$2, \$3, 12
  - I-type A/L/S/C instruction
  - Opcode is 001000, rs=3, rt=2, imm=12
  - 001000 00011 00010 0000000000001100

Example Instructions

- BEQ $3, $4, 4
  - I-type conditional branch instruction
  - Opcode is 000100, rs=00011, rt=00100, imm=4 (skips next 4 instructions)
  - 000100 00011 00100 0000000000000100

- SW $2, 128($3)
  - I-type memory address instruction
  - Opcode is 101011, rs=00011, rt=00010, imm=0000000010000000
  - 101011 00011 00010 0000000010000000

- J 128
  - J-type pseudodirect jump instruction
  - Opcode is 000010, 26-bit pseudodirect address is 128/4 = 32
  - 000010 00000000000000000000100000

MIPS – Some Load & Store Instructions

- Load Double Word

LD R1, 30(R2) ;$Regs[R1] \leftarrow_{64} Mem[30 + Regs[R2]]$

- Load Double Word (direct addressing using zero as base reg)

LD R1, 30(R0) ;$Regs[R1] \leftarrow_{64} Mem[30 + 0]$

- Load Byte

LB R1, 40(R3) ;$Regs[R1] \leftarrow_{64} (Mem[40 + Regs[R3]]_0)^{56}$ ## $Mem[40 + Regs[R3]]$

- Load Byte Unsigned

LBU R1, 40(R3) ;$Regs[R1] \leftarrow_{64} 0^{56}$ ## $Mem[40 + Regs[R3]]$

- Load FP single

L.S F0, 50(R3) ;$Regs[F0] \leftarrow_{64} Mem[50 + Regs[R3]]$ ## 0^{32}

- Load FP double

L.D F0, 50(R2) ;$Regs[F0] \leftarrow_{64} Mem[50 + Regs[R2]]$

- Store Half Word

SH R3, 502(R2) ;$Mem[502 + Regs[R2]] \leftarrow_{16} Regs[R3]_{48..63}$

- Store FP single

S.D F0, 40(R3) ;$Mem[40 + Regs[R3] \leftarrow_{32} Regs[F0]]_{0..31}$

MIPS – Some Arithmetic/Logic Instructions

- Double Word Add

DADD R1, R2, R3 ;Regs[R1] ← Regs[R2] + Regs[R3]

- Double Word Add Immediate Unsigned

DADDIU R1, R2, #3 ;Regs[R1] ← Regs[R2] + 3

- Double Word Shift Left Logical

DSLL R1, R2, #5 ;Regs[R1] ← Regs[R2] << 5

- Set on Less Than

SLT R1, R2, R3 ; if (Regs[R2] < Regs[R3])
Regs[R1] ← 1 else Regs[R1] ← 0

- Multiply and Add Word to HI,LO registers

MADD R1, R2 ; (LO,HI) ← Regs[R1] x Regs[R2] + (LO,HI)

- Loading a Constant (mnemonic: **LI**)

LI R1, #3 ;Regs[R1] ← 3

- Register-Register Move (mnemonic: **MOV**)

MOV R1,R2 ;Regs[R1] ← Regs[R2]

EXAMPLE


```
Loop:   add  $t1, $s3, $s3     # starts from 80000
        add  $t1, $t1, $t1
        add  $t1, $t1, $s6
        lw   $t0, 0($t1)
        bne  $t0, $s5, Exit
        add  $s3, $s3, $s4
        j    Loop
Exit:
```

EXAMPLE

	6	5	5	5	5	6	
80000	0	19	19	9	0	32	R-type
80004	0	9	9	9	0	32	R-type
80008	0	9	22	9	0	32	R-type
80012	35	9	8	0			I-type
80016	5	8	21	2			I-type
80020	0	19	20	19	0	32	R-type
80024	2	20000					J-type
80028							

MIPS – Some Arithmetic/Logic Instructions

- Suppose register R3 has the binary number

1111 1111 1111 1111 1111 1111 1111 1111

- and register R4 has the binary number

0000 0000 0000 0000 0000 0000 0000 0001

- What are the values of R1 and R2 after this?

SLT R1 R3 R4

SLTU R2 R3 R4

- Answer:

R1 = 1, R2 = 0

- because

R3 = -1 (as an integer) and =4,294,967,295 (as unsigned)

R4 = 1 in both cases

MIPS – Some Control-Flow Instructions

- Jump (28-bit target address is 26-bit name shifted left twice)

J name ;PC $\leftarrow$ PC$_{64..28}$ ## name ## 0^2

- Jump and link (using the return address register RA R31)

JAL name ;Regs[R31] $\leftarrow$ PC+8; PC $\leftarrow$ PC$_{64..28}$ ## name ## 0^2

- Jump and link register

JALR R2 ;Regs[R31] $\leftarrow$ PC+8; PC $\leftarrow$ Regs[R2]

- Jump register

JR R3 ;PC $\leftarrow$ Regs[R3]

- Branch equal (zero) (compilers combined this with SLT, SLTI in order to build branch-on-less conditions)

BEQ R4, R0, name ;if (Regs[R4]==0) PC $\leftarrow$ PC + signExt{name ## 0^2 }

- Branch not equal

BNE R3,R4,name ;if(Regs[R3]!= Regs[R4]) PC $\leftarrow$ PC + signExt{name ## 0^2 }

- Conditional move if zero (support conversion of a simple branch into a conditional arithmetic instruction)

MOVZ R1,R2,R3 ;if(Regs[R3]==0) Regs[R1] $\leftarrow$ Regs[R2]

Review: MIPS-Jump Instruction Example


```
/* Assume loop stored at
   0000 A000 6BC4 0308 */
Loop:  SLL R9,R19,2
       ADD R9,R9,R22
       LW  R8, 0(R9)
       BNE R8, R21,exit
       ADDI R19,R19,1
       J Loop
Exit:  LW R11, 0(R18)
```

- `j loop` : [PC-region addressing] the target address effectively encoded is `2F9 00C2` that shifted-left by two and composed with the upper 36 bits of the PC of the following instruction gives `0000 a000 6bc4 0308`
- `BNE R8, R21, exit` : [PC-relative addressing] the offset is 2 (it gets summed to the PC of the following instruction to point to Exit)

0000 a000 6bc4 0308 :	SLL	0	R19	R9	2	sll
0000 a000 6bc4 030C :	ADD	R9	R22	R9	000000	add
0000 a000 6bc4 0310 :	LW	R9	R8	0		
0000 a000 6bc4 0314 :	BNE	R8	R21	2		
0000 a000 6bc4 0318 :	ADDI	R19	R19	1		
0000 a000 6bc4 031C :	JUMP	2F9 00C2 = (bc4 0308 / 4)				
0000 a000 6bc4 0320 :	LW	R18	R11	0		

Review: MIPS-Jump Instruction Example


```
/* Assume loop stored at
   0000 A000 6BC4 0308 */
Loop: SLL R9,R19,2
      ADD R9,R9,R22
      LW  R8, 0(R9)
      BNE R8, R21,exit
      ADDI R18,R18,1
      J Loop
Exit: LW R11, 0(R18)
```

- `j loop` : [PC-region addressing] the target address effectively encoded is `2F9 00C2` that shifted-left by two and composed with the upper 36 bits of the PC of the following instruction gives `0000 a000 6bc4 0308`
- `BNE R8, R21, exit` : [PC-relative addressing] the offset is 2 (it gets summed to the PC of the following instruction to point to Exit)

0000 a000 6bc4 0308 :	000000	000000	010011	001001	000010	000000
0000 a000 6bc4 030C :	000000	001001	010110	001001	000000	100000
0000 a000 6bc4 0310 :	100011	001001	001000	000000		
0000 a000 6bc4 0314 :	000101	001000	010101	000010		
0000 a000 6bc4 0318 :	001000	010011	010011	000001		
0000 a000 6bc4 031C :	000010	1011 1110 0100 0000 0011 0000 10				
0000 a000 6bc4 0320 :	100011	010010	001011	000000		

MIPS – Floating-Point Instructions

- Floating Point Absolute Value (Double Precision)

ABS.D F1, F2 ;F1 ← abs(F2)

- Floating Point Add (Single Precision)

ADD.S F3, F1, F2 ;Regs[F3] ← Regs[F1] + Regs[F2]

- Floating Point Add (Double Precision)

ADD.D F3, F1, F2 ;Regs[F3] ← Regs[F1] + Regs[F2]

- Floating Point Divide (Double Precision)

DIV.D F3, F1, F2 ;Regs[F3] ← Regs[F1] / Regs[F2]

- Floating Point Square Root (Double Precision)

SQRT.D F3, F1 ;Regs[F3] ← sqrt(Regs[F1])

- Floating Point Multiply Add (Single Precision)

MADD.S F3, F1, F2, F4 ;Regs[F3] ←(Regs[F2] x Regs[F4)] +
+ Regs[F1]

Real World Instruction Sets

Arch	Type	# Oper	# Mem	Data Size	# Regs	Addr Size	Use
Alpha	Reg-Reg	3	0	64-bit	32	64-bit	Workstation
ARM	Reg-Reg	3	0	32/64-bit	16	32/64-bit	Cell Phones, Embedded
MIPS	Reg-Reg	3	0	32/64-bit	32	32/64-bit	Workstation, Embedded
SPARC	Reg-Reg	3	0	32/64-bit	24-32	32/64-bit	Workstation
TI C6000	Reg-Reg	3	0	32-bit	32	32-bit	DSP
IBM 360	Reg-Mem	2	1	32-bit	16	24/31/64	Mainframe
x86	Reg-Mem	2	1	8/16/32/ 64-bit	4/8/24	16/32/64	Personal Computers
VAX	Mem-Mem	3	3	32-bit	16	32-bit	Minicomputer
Mot. 6800	Accum.	1	1/2	8-bit	0	16-bit	Microcontroler

Why the Diversity in ISAs?

Technology Influenced ISA

- Storage is expensive, tight encoding important
- Reduced Instruction Set Computer
  - Remove instructions until whole computer fits on die
- Multicore/Manycore
  - Transistors not turning into sequential performance

Application Influenced ISA

- Instrucions for Applications
  - DSP instructions
- Compiler Technology has improved
  - SPARC Register Windows no longer needed
  - Compiler can register allocate effectively

Quize

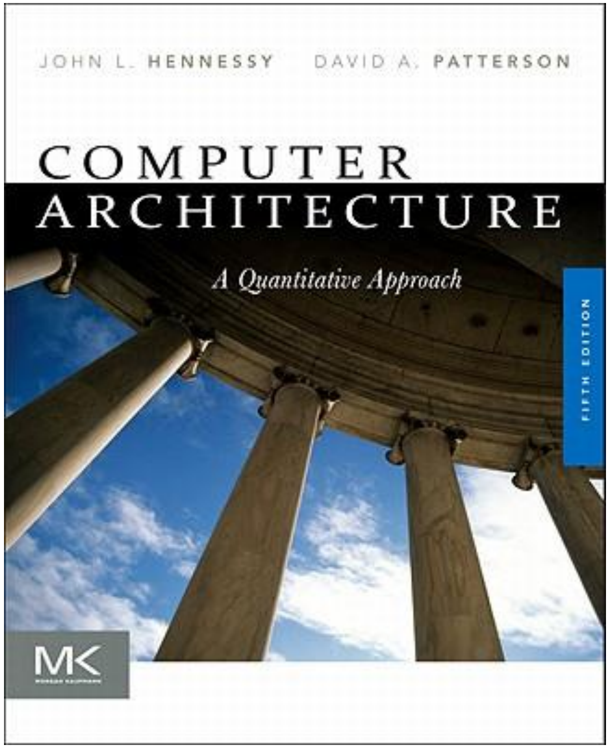
1. H&P5th Page A-47 A.1

2.
[Self-Study]：
The value represented 0x(0123 4567 89AB CDEF) is to be stored in an aligned 64bit double word. Choose an address from 0x00002 to 0x00012 as the starting address, and store them using Little Endian byte order.

Assigned Readings

- Computer Architecture: A Quantitative Approach, 5th ed.," John L. Hennessey and David A. Patterson, Morgan Kaufman, 2011

- Sections: 1.1-1.6, Appendix L



Further Readings

- Martin Davis. “Engine of Logic”. Norton, 2000
- Hermann H. Goldstine. “The Computer: From Pascal to Von Neumann”. Princeton University Press, 1972
- Andrew Hodges. “Alan Turing: The Enigma”. Walker & Company, 2000
- Eloina Pelaez. “The Stored-Program Computer: Two Conceptions”. In Social Studies of Science 29(3), June 1999.
- Computer History Museum. http://computerhistory.org