McGill

Chapter 2

Instruction Set Architectures

ECSE 425
Winter 2007

Review

- Patterson and Hennesy: Computer Organization and Design: The Hardware/Software Interface (2’nd ed.)
  - Chapter 3

- We will focus in the lecture **on the most important concepts** as they apply to RISC machines. Read chapter 2 of the course text.

Instruction Set Architecture (ISA)

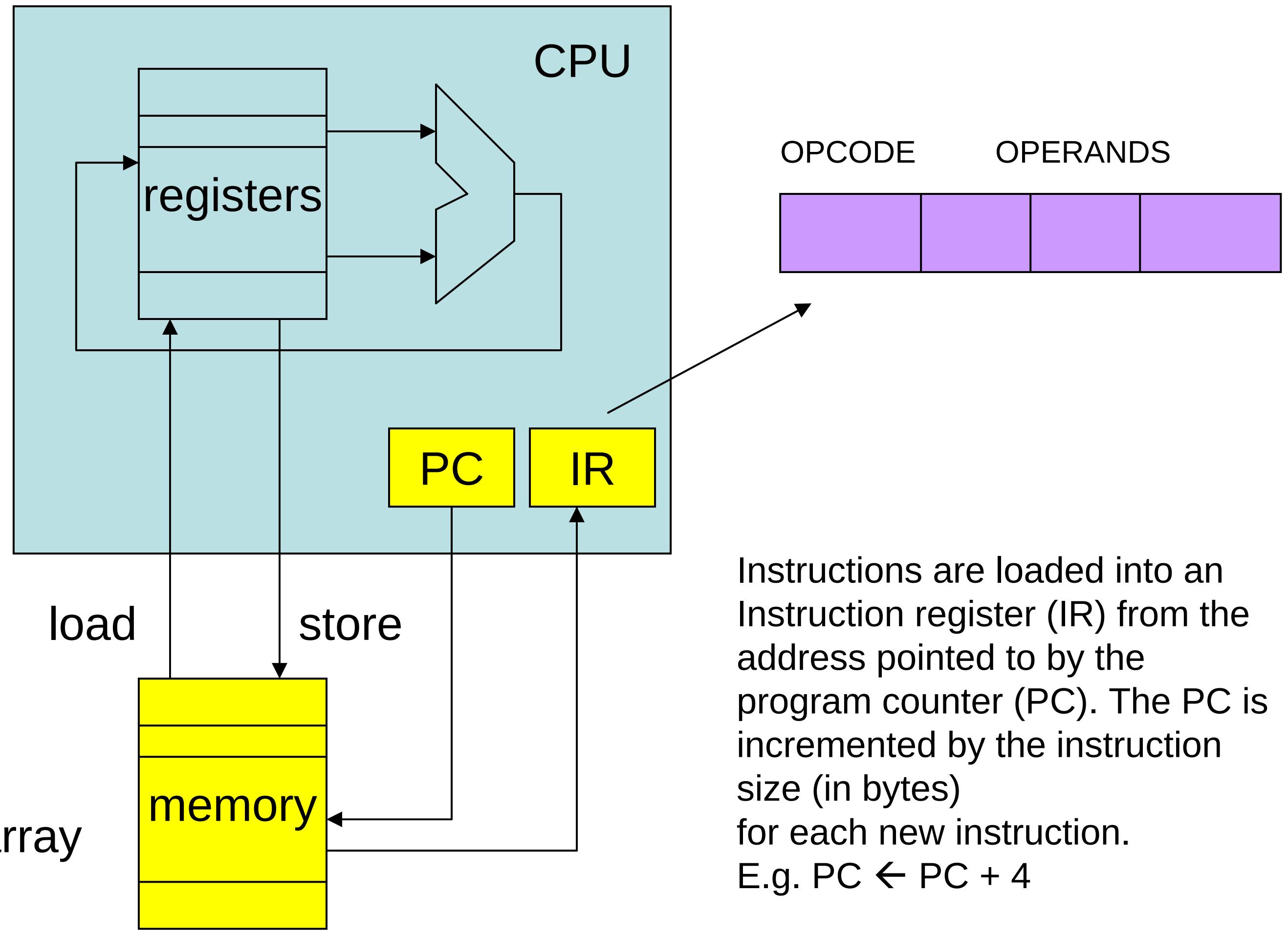
- Computers run programs made of simple operations called “instructions”
- The list of instructions offered by the machine is the “instruction set”
- The instruction set is what is visible to the programmer (really the compiler, although humans can directly program in “assembly language”).

Instructions

- Two kinds of information in a computer:
  - instructions
  - data
- Instructions are stored as numbers, just like data
- Instructions and data are stored in the memory

Basic Computer Organization

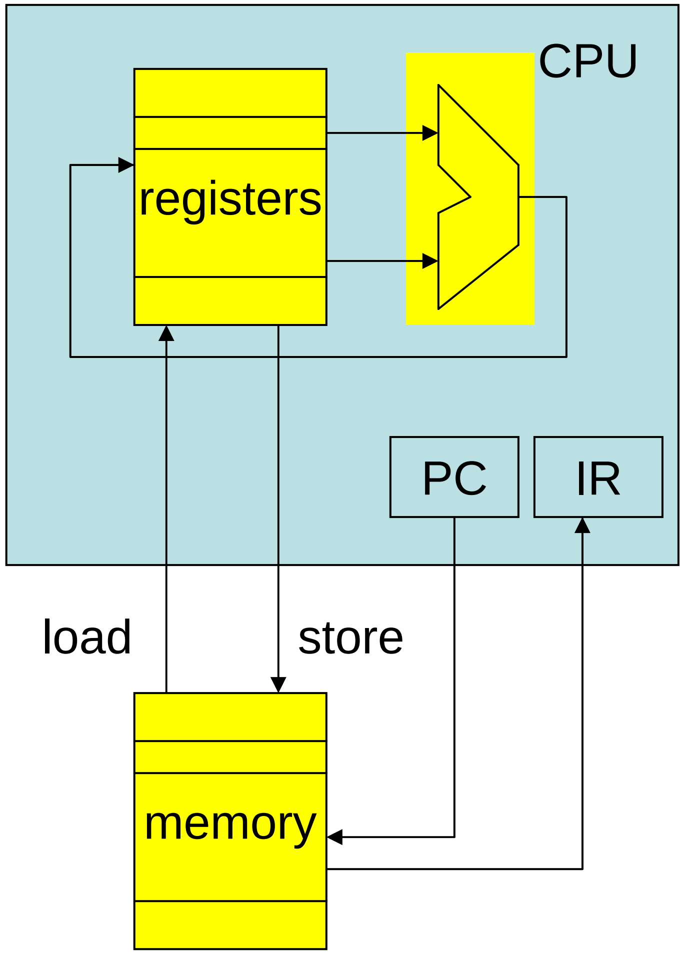
Limited number of **fast** registers for temporary storage



Large amount of **slow** memory Arranged as an array of bytes

Instructions are loaded into an Instruction register (IR) from the address pointed to by the program counter (PC). The PC is incremented by the instruction size (in bytes) for each new instruction.
E.g. PC ← PC + 4

Load/Store Architecture (Reg-Reg)

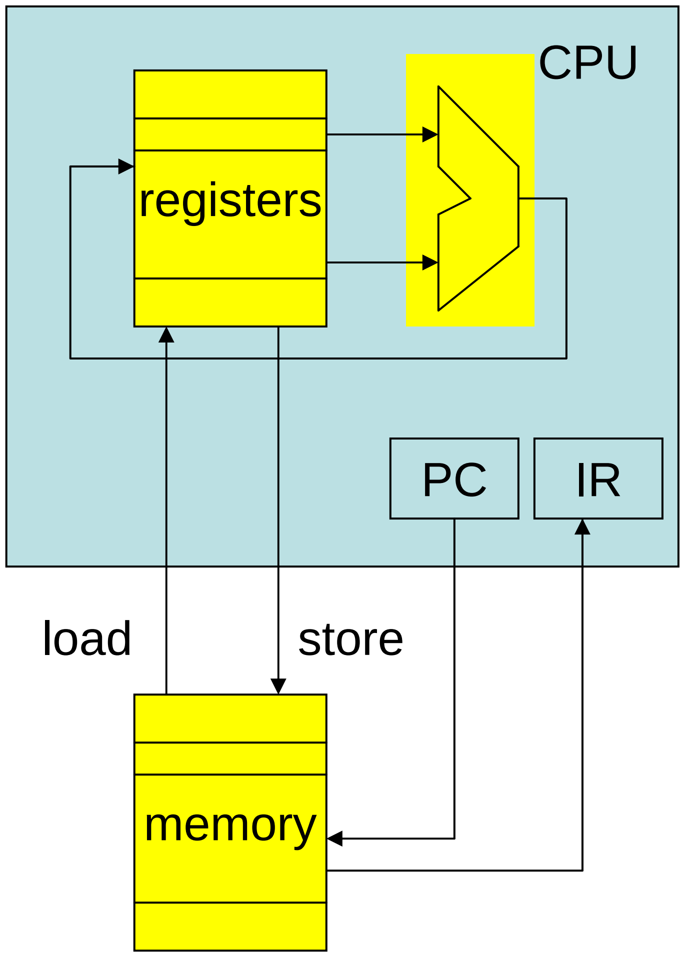


• Instructions can **ONLY** get their data and store their data from/to registers.

• The register numbers are specified in the operand fields of the instruction

• Since data is stored in memory, we need special “load” and “store” instructions for transfers between registers and memory. These two instructions are the ONLY ones allowed to access memory

Load/Store Architecture (Reg-Reg)



• **RISC** architectures are load/store. The regularity of this architecture enables fast organizations using pipelining (next chapter).

• **CISC** machines (e.g. Intel IA-32) permit instructions to get their data from both registers and memory (mem-reg). These highly irregular architectures (mem-reg, variable-length instructions) are practically impossible to pipeline.

• The advantage of them is that they produce shorter programs (no loads or stores needed, variable-length instr.d), but memory today is cheap and compilers can't really use complex instructions anyways.

• Modern "CISC" machines really just translate the CISC instructions to a set of RISC instructions and run those.
  - done purely for compatability reasons.

Other ISAs

- Some old ISAs use a small number of special-purpose registers arranged as a stack or accumulator (single register).
- These special-purpose registers constrain compilers.
- Compilers like flexibility !
- ISAs should have lots of general-purpose registers.

Example

(A * B) + (C * D) (high – level language)

Machine instructions (assembly language)

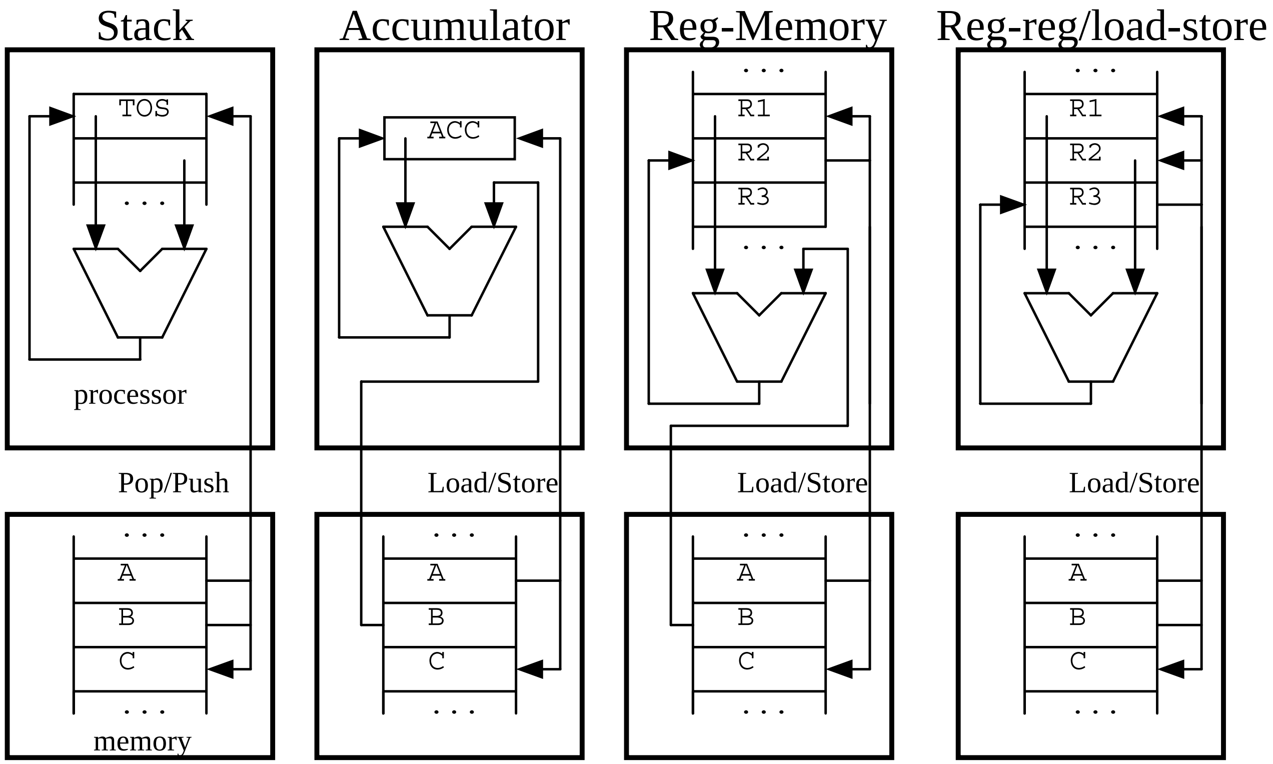
```
LOAD  R1, A
LOAD  R2, B
MUL   R3, R1, R2
LOAD  R1, C
LOAD  R2, D
MUL   R4, R1, R2
ADD   R5, R3, R4
STORE R5, RESULT
```

Classification of ISAs

- Implicit operand(s):
  - **STACK**: operands implicit (e.g. ADD).
    - TOS: pointer to top of the stack.
    - PUSH items onto stack. Items PUSH/POP'ed off – LIFO.
    - ex. reverse polish notation calculator
  - **ACCUMULATOR**: one operand implicitly "accumulator" register
    - Ex. ADD B (to contents of accumulator). Result goes to accumulator register implicitly.

- Explicit operands, register or memory:
  - **REGISTER-MEMORY**
    - access memory as part of any instruction.
  - **REGISTER-REGISTER (or LOAD/STORE)**
    - can only access memory with loads and stores.
  - **MEMORY-MEMORY**
    - keeps all operands in memory (not found much today)

C = A + B



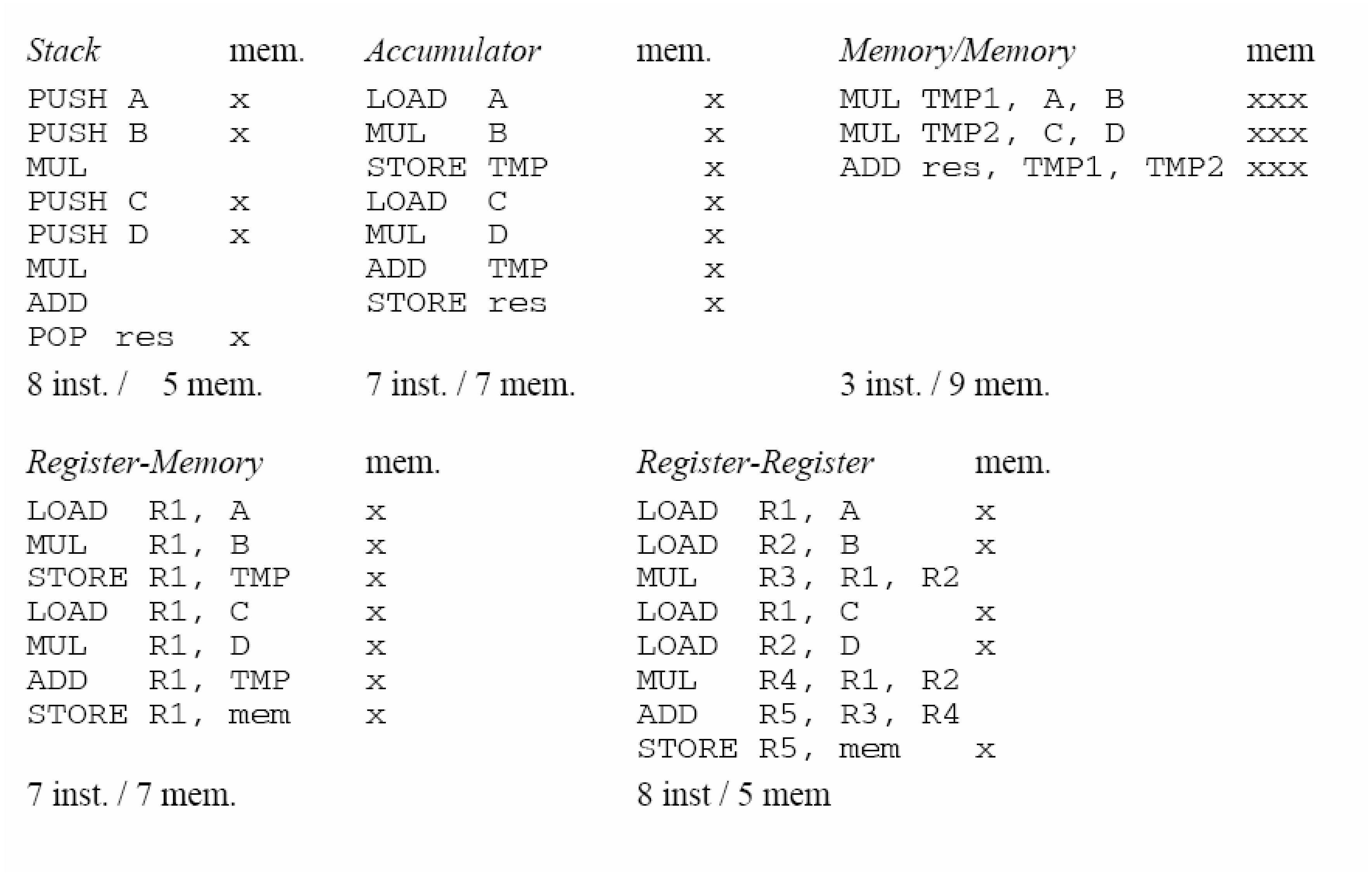
```
Push A          Load  A          Load  R1,A         Load  R1,A
Push B          Add   B          Add   R2,R1,B      Load  R2,B
Add             Store C          Store R2,C         Add   R3,R1,R2
Pop  C                                              Store R3,C
```

Load-Store Architectures

- Early computers used a stack or accumulator.
- Since 1980, virtually all LOAD-STORE.
  - Registers faster than memory (internal to processor)
  - More efficient for compilers – can perform operation in any order.
    - ex. (A x B) + (C x D). Stack has to be in order.
  - Registers can hold variables
    - reduced memory traffic,
    - Faster programs,
    - code density improves (register fewer bits than memory).

Code Template	# of memory addresses	Max # of operands	Type of architecture	Example
Push A Push B Add Pop C	0	0	Stack	Almost extinct
Load A Add B Store C	1	1	Accumulator	Almost extinct
Add C,A,B	3	3	Memory/Memory	VAX
Add B,A	2	2	Memory/Memory	VAX
Load R1,A Add R1,B Store R1,C	1	2	Register/Memory	IBM360, 80x86, 68000, TMS320C54x
Load R1,A Load R2,B Add R3,R1,R2 Store R3,C	0	3	Register/Register or Load/Store.	Alpha, ARM, MIPS, PowerPC, SPARC, SuperH, Trimedia

Type	Advantages	Disadvantages
Stack (0,0)	Very small instructions (pocket calculators!)	Lots of memory traffic
Accumulator (1,1)	Simple instructions. (Very simple pic proc.)	One single register: memory traffic.
Memory/Memory (2,2)(3,3)	Most compact (good instr. encoding) Efficient use of the registers	Instructions vary in *length* and *work performed* (CPI). Makes pipelining imposs. Bottleneck! No longer used, legacy only.
Register-Memory (1,2)	Needs one load only. Good encoding. Good density.	Destroys (re-writes) one source operand in 2 operand case. Number of registers limited. CPI varies, making pipelining hard.
Register-Register (0,3)	Simple fixed length. Easy to compile for. Uniform CPI. (see App. A)	Higher instr. count. Lower density makes large object code. (but memory cheap)



Stack mem.

```
PUSH A    x
PUSH B    x
MUL
PUSH C    x
PUSH D    x
MUL
ADD
POP  res  x
```

8 inst. / 5 mem.

Accumulator mem.

```
LOAD   A     x
MUL    B     x
STORE  TMP   x
LOAD   C     x
MUL    D     x
ADD    TMP   x
STORE  res   x
```

7 inst. / 7 mem.

Memory/Memory mem

```
MUL TMP1, A, B        xxx
MUL TMP2, C, D        xxx
ADD res, TMP1, TMP2   xxx
```

3 inst. / 9 mem.

Register-Memory mem.

```
LOAD  R1, A     x
MUL   R1, B     x
STORE R1, TMP   x
LOAD  R1, C     x
MUL   R1, D     x
ADD   R1, TMP   x
STORE R1, mem   x
```

7 inst. / 7 mem.

Register-Register mem.

```
LOAD  R1, A       x
LOAD  R2, B       x
MUL   R3, R1, R2
LOAD  R1, C       x
LOAD  R2, D       x
MUL   R4, R1, R2
ADD   R5, R3, R4
STORE R5, mem     x
```

8 inst / 5 mem

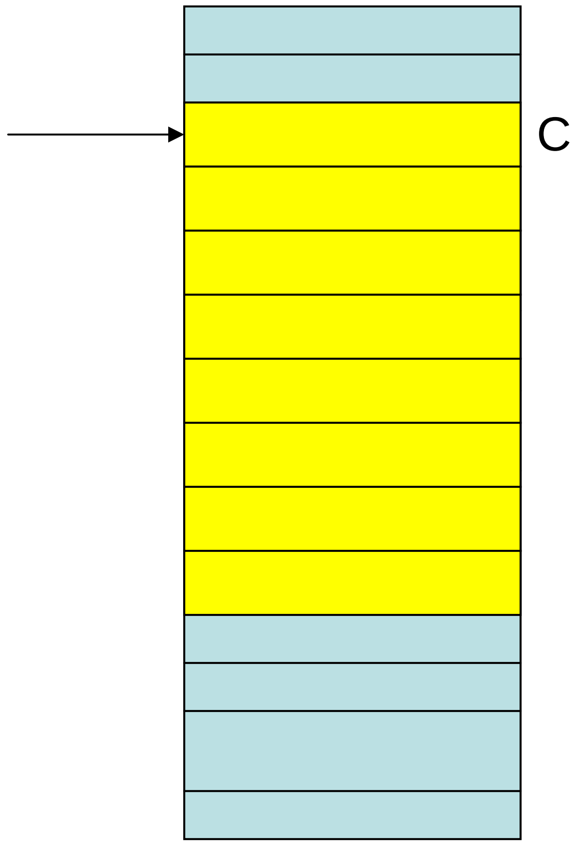
Data Types

- Integer
  - 8 bits (char)
  - 16-bits (short or half-word)
  - 32-bits (word)
  - 64 bits (double word)
- Floating point
  - single-precision (32-bits)
  - double-precision (64-bits)

Memory Addressing

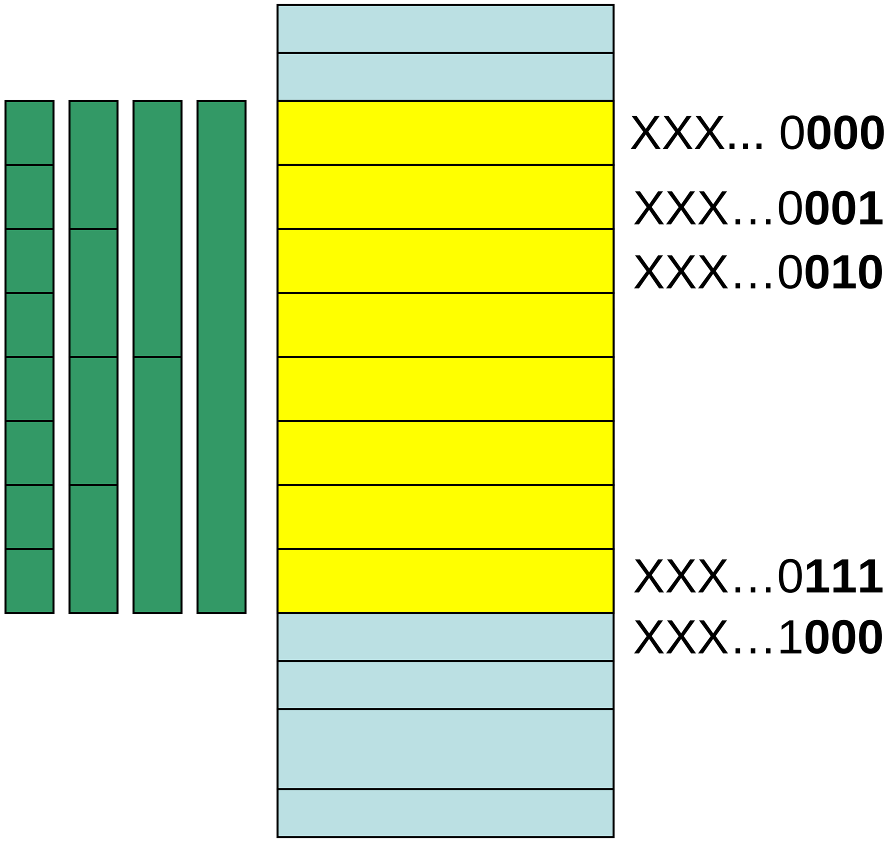
- Each byte (8-bits) in the memory is given a unique address.
- Data can be accessed in chunks of multiple bytes by giving the address of the starting byte and the size of the chunk

- E.g. LD R4, C
- Loads a “double word” (8 bytes) starting at address C

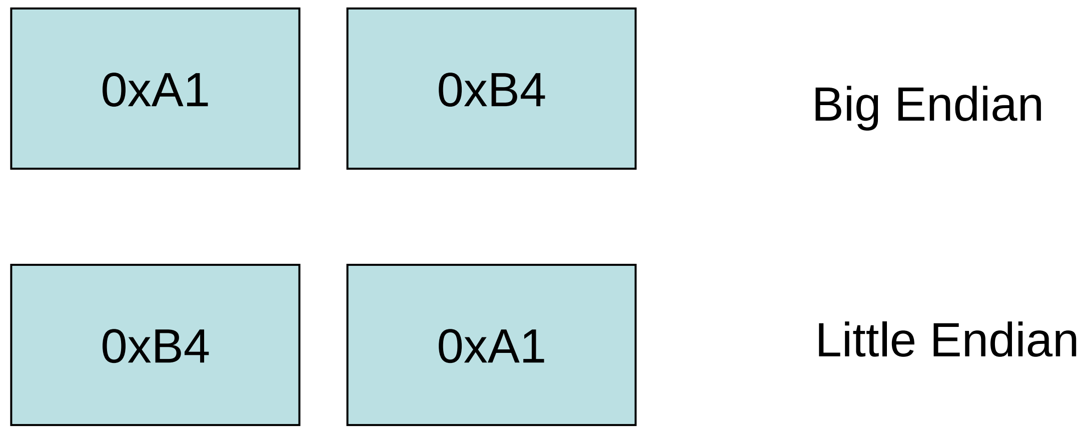


Alignment

- Some computers require the memory access must start on an address that is a multiple of the chunk size in bytes (i.e. half-words can only be accessed on bytes 0, 2, 4, 6, …)



Byte Ordering



Addressing Modes

- How to specify the “effective address” to an instruction

- Register-Transfer Language (Chapter 2 of the textbook). We will do some examples. Read Ch. 2 and especially Figure 2.6 to see more examples

Addressing Modes

- Immediate (constants)

```
Add R4, #3                 Regs[R4] ← Regs[R4] + 3
```

- Register-Register

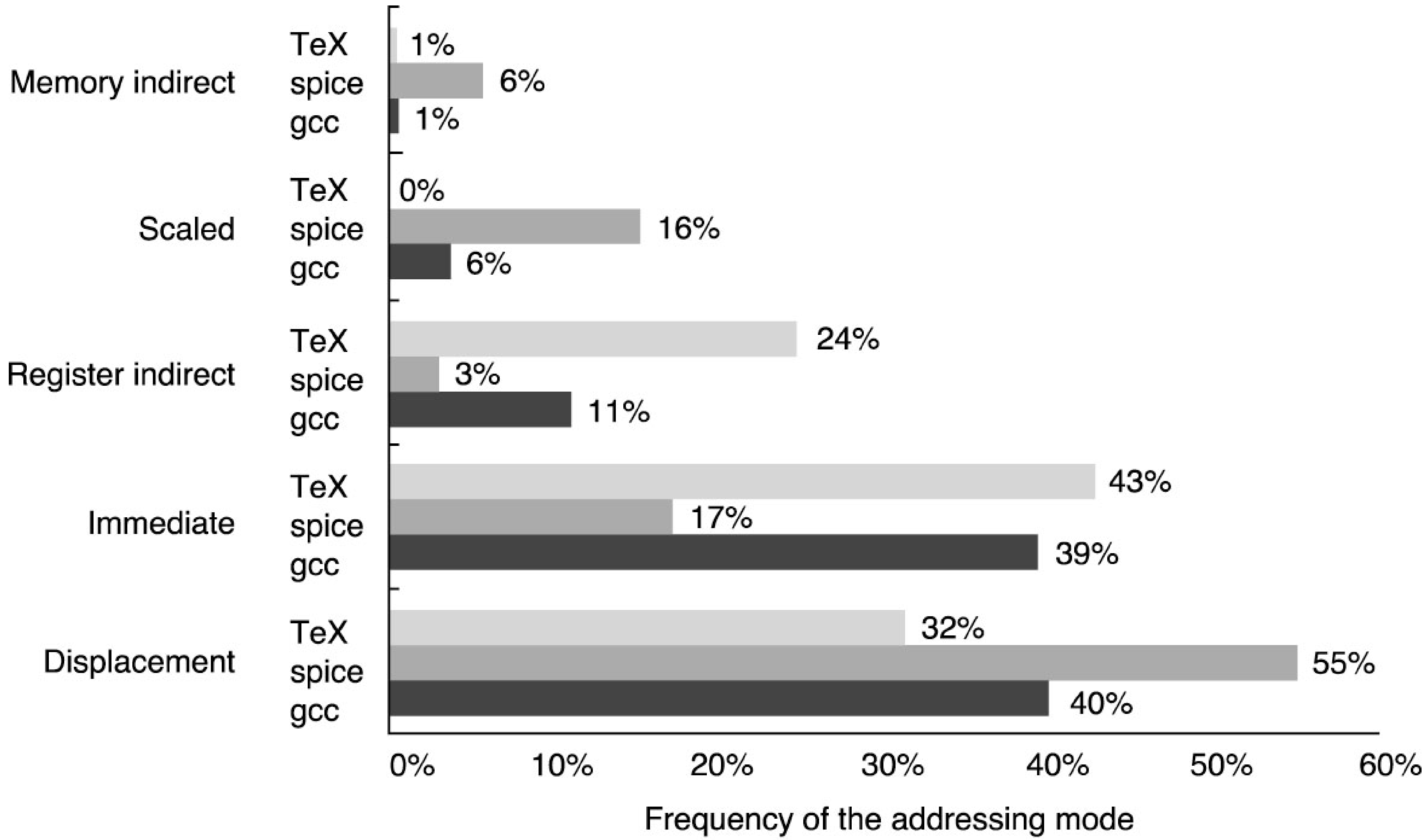
```
Add R4, R5, R6             Regs[R4] ← Regs[R5] + Regs[R6]
```

- Displacement (computed addresses, pointers, local variables array accesses)

```
Load R4, 100(R3)           Regs[R4] ← Mem[Regs[R3] + 100]
```

- For others (not really used in RISC too often), see Figure 2.6.

Name	Example	RTL	When used
Register Indirect	Add R4,(R1)	Regs[R4]←Regs[R4]+Mem[Regs[R1]]	Pointer access or Computed addresses.
Indexed	Add R4,(R1+R2)	Regs[R4]←Regs[R4]+Mem[Regs[R1]+Regs[R2]	Array addressing
Absolute	Add R4,(1001)	Regs[R4]←Regs[R4]+Mem[1001]	Static data access.
Memory Indirect	Add R4,@(R1)	Regs[R4]←Regs[R4]+ Mem[Mem[Regs[R1]]	*p when &p is in reg R1
Autoincrement	Add R4,(R2)+	Regs[R4]←Regs[R4]+Mem[Regs[R2]] Regs[R2]←Regs[R2]+d	Array stepping. Stack access.
Autodecrement	Add R4,-(R1)	Regs[R2]←Regs[R2]-d Regs[R4]←Regs[R4]+Mem[Regs[R2]]	Array stepping. Stack access.
Scaled	Add R4,100(R1,R2)	Regs[R4]←Regs[R4]+ Mem[100+Regs[R1]+Regs[R2]*d]	Arrays


Memory indirect
Scaled
Register indirect
Immediate
Displacement
TeX
spice
gcc
1%
6%
1%
0%
16%
6%
24%
3%
11%
43%
17%
39%
32%
55%
40%
0%
10%
20%
30%
40%
50%
60%
Frequency of the addressing mode

Operations

Arithmetic and Logical	Add, subtract (signed, unsigned), and, or, shifts, multiply, divide.
Data Transfer	Load, Store
Control	Branch, jump, procedure call, return, trap.
System	Syscall, Virtual memory management
Floating Point	FPadd, FPmult, FPdiv, FPcompare
Decimal	Arithmetic and conversion
Strings	Move, copy, compare, search
Graphics	Pixel, Vertex ops, compress, decompress

Operations

- It is often the case that few instruction statistically dominate.
  - e.g. SPEC92 benchmark indicates (80x86):

Loads:	**22%**
Branches:	**20%**
Compare:	**16%**
Store:	**12%**
ALU:	**19%**

- Important conclusions:
  - 5 (simple) types make 89% of all instructions
    - make these fast!
  - twice as many loads than stores (more reads than writes

Control Flow (Branch)

- How to change to flow of a program

BEQ, BNE, BEQZ, BNEZ, etc.…

```
if (x != y)
  instruction_a

instruction_b
```

`(x stored in R1, y stored in R2)`

```
       BEQ R1, R2, label
        instruction_a
label:  instruction_b
```

Address of the instruction in memory to execute if the condition is true (**target**), else, fall through to the next sequential instruction

BEQZ R1, name

condition

• tradeoff : how many bits to allocate in the instruction field for the target address (displacement) (PC ← PC + displacement)

Subcategories

- Branches (dominant)
- Jumps
- Procedure calls
- Procedure return

Target Addressing Modes

- PC-relative
  - add an offset to current PC (program position independent).

- Register indirect
  - target in a register
    - Procedure returns
    - Case or switch statements
    - Virtual function or methods (pick procedure according to args)
    - Function pointers (pass a function as an argument)
    - DSL's (Dynamically Shared Libraries, e.g. DLLs, Unix modules)

  - In these cases, the target address is not known at compile time, nor at link time, it is computed on the fly.

Branch Options

Name	Examples	Test	Advantage	Disadvantage
Condition Code	80x86 ARM PowerPC Sparc	Special register CC set by ALU fspossibly under program control	Condition can be set at no cost	CC is extra state. Constrains ordering of instructions.
Condition register	Alpha, MIPS	Use any GP register to store result of a comparison.	Simple, regular.	Use a register for 1 bit.
Compare and branch	PA-RISC, VAX	Two instructions packed in one.	High level. 1 instr./branch	Hard to pipeline.

Instruction Encoding

- **How is the ISA encoded in binary strings (machine code)?**
- **opcode, followed by operand encoding.**
  - **Operand encoding (and hence instruction decoding) becomes more complex as the number of supported addressing modes increase. (RISC-CISC argument).**
- **fixed length**
  - **fixed number of operands**
  - **combines operation and addressing mode into opcode**
  - **Fixed instruction length, larger code representaion, easy to decode.**
- **variable length**
  - **any number of operands, permits all addressing modes**
  - **Flexible instruction length, smaller code representation, harder to decode.**
- **Encodings that make possible pipelining and advanced pipelining.**

Compilers

- Can do program transformations (optimization) to improve performance
- Whatever it does, the resulting transformed program must be correct
- Read more about this in text

Compilers

- Increased **role of compilers** in system design
  - balance the job of the hardware and that of the software
  - These are no longer separate problems
- Goals:
  - All correct programs compile/work correctly
  - Most compiled programs execute quickly
  - Most programs compile quickly
  - Achieve small code size
  - Provide debugging support

Dependencies	Stages (phase)	Function
High-Level language dependent.	Front end	Transform language into common intermediate form.
Some language dependencies	High-level Optimizations	E.g. Loop transformations, procedure in-lining, dead-code elimination, constant folding,…
Small dependencies on language and on target machine, *e.g.* number of GPRs.	Global optimizer	Global and local optimizations. Register allocation (NP-complete problem: heuristic). Common sub expression elimination. Constant propagation. Stack height reduction. Copy propagation Code motion Eliminate array addressing.
Highly machine dependent.	Code generator.	Detailed instruction selection and machine dependent optimization: Peephole optimizations (many), strength reduction, pipeline scheduling, Branch optimization (many)

Optimizations

- **Local:**
  - **Common subexpr elimination**: Replace expressions that compute the same result
  - **Constant propagation**: Replace all instances of a variable that is assigned a constant with the constant
  - **Stack height reduction**: Rearrange expression to minimize resources (stack) needed for expression evaluation

- **Global:**
  - **Global common subexpression elimination**: Same as local, but across branches
  - **Copy propagation**: Replace all instances of a variable that has been assigned
  - **Code motion** : Remove code from loop that computes the same value for each iteration of loop.
  - **Eliminate array addressing (global):** simplify/eliminate array addressing calculations within loops

Example

- **Eliminating common sub-expressions/strength reduction**

y = A + B * x + C * (x2) + D * (x**3)   (original code)**

- The following forms are more efficient to compute because they require fewer and ‘lighter’ operations.
  - Stage #1: **y = A + (B + C * x + D * (x**2)) * x**
  - Stage #2: **y = A + (B + (C + D * x) * x) * x**
  - The last form requires 3 additions and only 3 multiplications!

Impact of ISA on Compiler

- "Make the frequent case fast and the rare case correct"!
- Instruction set properties to help the compiler writer:
  - Provide regularity: *Orthogonal* architecture: All operations, data types, addressing modes independent – e.g. every operation applies to all addressing modes.
  - Provide clear *primitives,* not linked to language idiosyncrasies.
- Special instructions: Media support (MMX Streaming SIMD)

MIPS-64 ISA

- Will be used for rest of the course
- A classic RISC ISA
- We will just cover the highlights. See textbook ch. 2, appendix C for details.

Registers

- 32 64-bit GPRs
  - R0, … R31
  - R0 is hardwired to zero (and writing to it does nothing)

- 32 FP regs
  - F0, … F31

- Special regs: e.g. FP condition codes

Data Types

- 8-bit bytes
- 16-bit half-words
- 32-bit words
- 64-bit double words

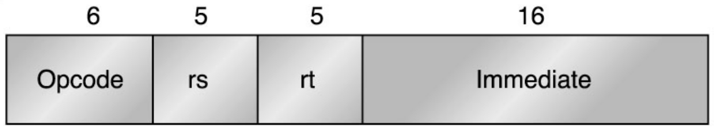
- 32-bit single-precision FP
- 64-bit double-precision FP

Addressing

- Addressing Modes
  - Immediate (16-bit)
  - Displacement (16-bit)
  - Can simulate other modes using R0
- Byte addressable
- 64-bit addresses
- Aligned accesses

Instruction Format

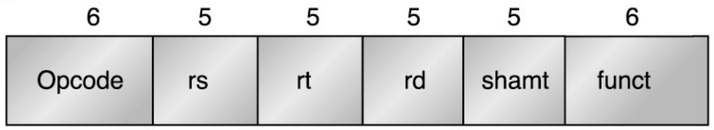
I-type instruction



Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt ← rs op immediate)

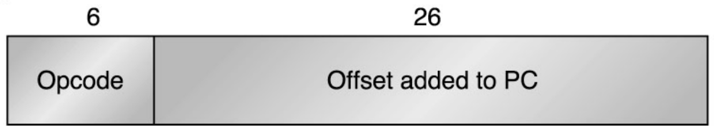
Conditional branch instructions (rs is reg1, rd reg2)
Jump register, jump and link register
(rd = 0, rs = destination, immediate = 0)

R-type instruction



Register-register ALU operations: rd ← rs funct rt
Function encodes the data path operation: Add, Sub, . . .
Read/write special registers and moves

J-type instruction



Jump and jump and link
Trap and return from exception

Operations

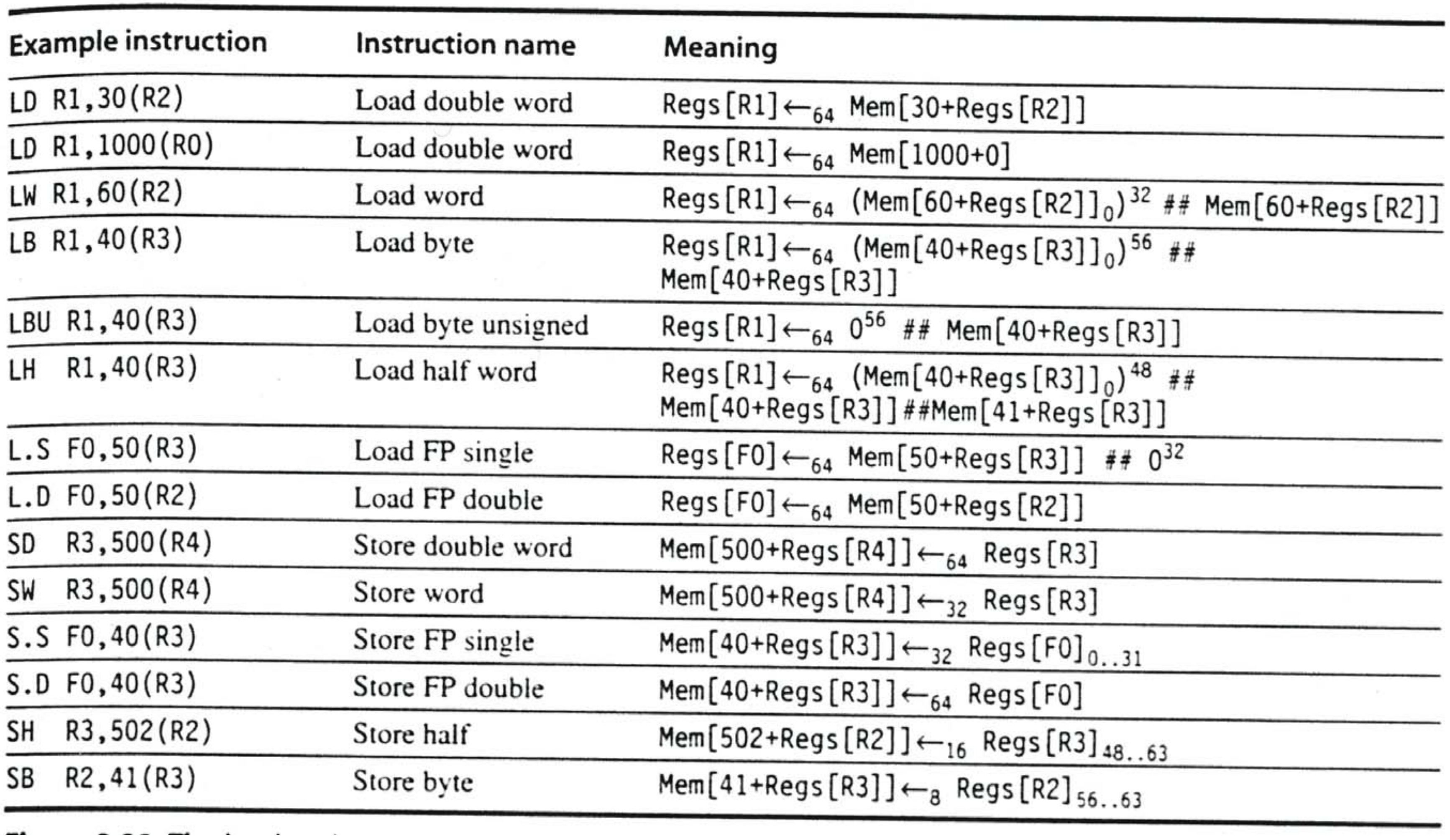
- Load/Stores
- ALU
- Branches and Jumps
- FP operations

- More on RTL.
  - $\leftarrow_n$ transfer *n* bits, *x,y* $\leftarrow$ *z* means transfer to *x* and *y*.
  - Subscript on quantity means bit selection (like an array of bits)
  - Regs[R4]_0 means sign bit of R4, $\text{Regs[R4]}_{56\ldots63}$means least significant byte.
  - Mem is an array of bytes
  - Superscript replicates field. 0^{48} is a field of 48 zeros.
  - ## concatenates fields.

- Example: byte at memory location addressed by the contents of register R8 is sign extended to form a 32-bit quantity that is stored in the lower half of register R10 (the upper half of R10 is unchanged)

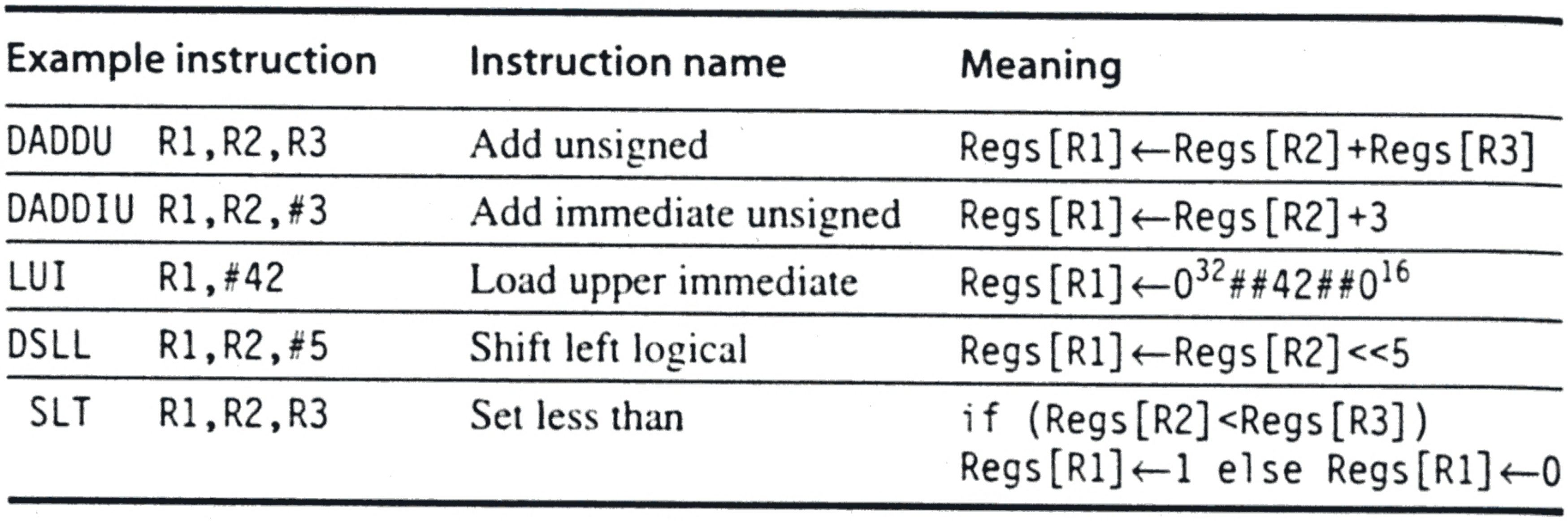
$$\texttt{Regs[R10]}_{32\ldots63} \leftarrow_{32} (\texttt{Mem[Regs[R8]]}_0)^{24} \; \texttt{\#\#} \; \texttt{Mem[Regs[R8]]}$$

Load / Stores



Example instruction	Instruction name	Meaning
LD R1,30(R2)	Load double word	Regs[R1] $\leftarrow_{64}$ Mem[30+Regs[R2]]
LD R1,1000(R0)	Load double word	Regs[R1] $\leftarrow_{64}$ Mem[1000+0]
LW R1,60(R2)	Load word	Regs[R1] $\leftarrow_{64}$ (Mem[60+Regs[R2]]$_0$)^{32} ## Mem[60+Regs[R2]]
LB R1,40(R3)	Load byte	Regs[R1] $\leftarrow_{64}$ (Mem[40+Regs[R3]]$_0$)^{56} ## Mem[40+Regs[R3]]
LBU R1,40(R3)	Load byte unsigned	Regs[R1] $\leftarrow_{64}$ 0^{56} ## Mem[40+Regs[R3]]
LH R1,40(R3)	Load half word	Regs[R1] $\leftarrow_{64}$ (Mem[40+Regs[R3]]$_0$)^{48} ## Mem[40+Regs[R3]]##Mem[41+Regs[R3]]
L.S F0,50(R3)	Load FP single	Regs[F0] $\leftarrow_{64}$ Mem[50+Regs[R3]] ## 0^{32}
L.D F0,50(R2)	Load FP double	Regs[F0] $\leftarrow_{64}$ Mem[50+Regs[R2]]
SD R3,500(R4)	Store double word	Mem[500+Regs[R4]] $\leftarrow_{64}$ Regs[R3]
SW R3,500(R4)	Store word	Mem[500+Regs[R4]] $\leftarrow_{32}$ Regs[R3]
S.S F0,40(R3)	Store FP single	Mem[40+Regs[R3]] $\leftarrow_{32}$ Regs[F0]$_{0..31}$
S.D F0,40(R3)	Store FP double	Mem[40+Regs[R3]] $\leftarrow_{64}$ Regs[F0]
SH R3,502(R2)	Store half	Mem[502+Regs[R2]] $\leftarrow_{16}$ Regs[R3]$_{48..63}$
SB R2,41(R3)	Store byte	Mem[41+Regs[R3]] $\leftarrow_{8}$ Regs[R2]$_{56..63}$

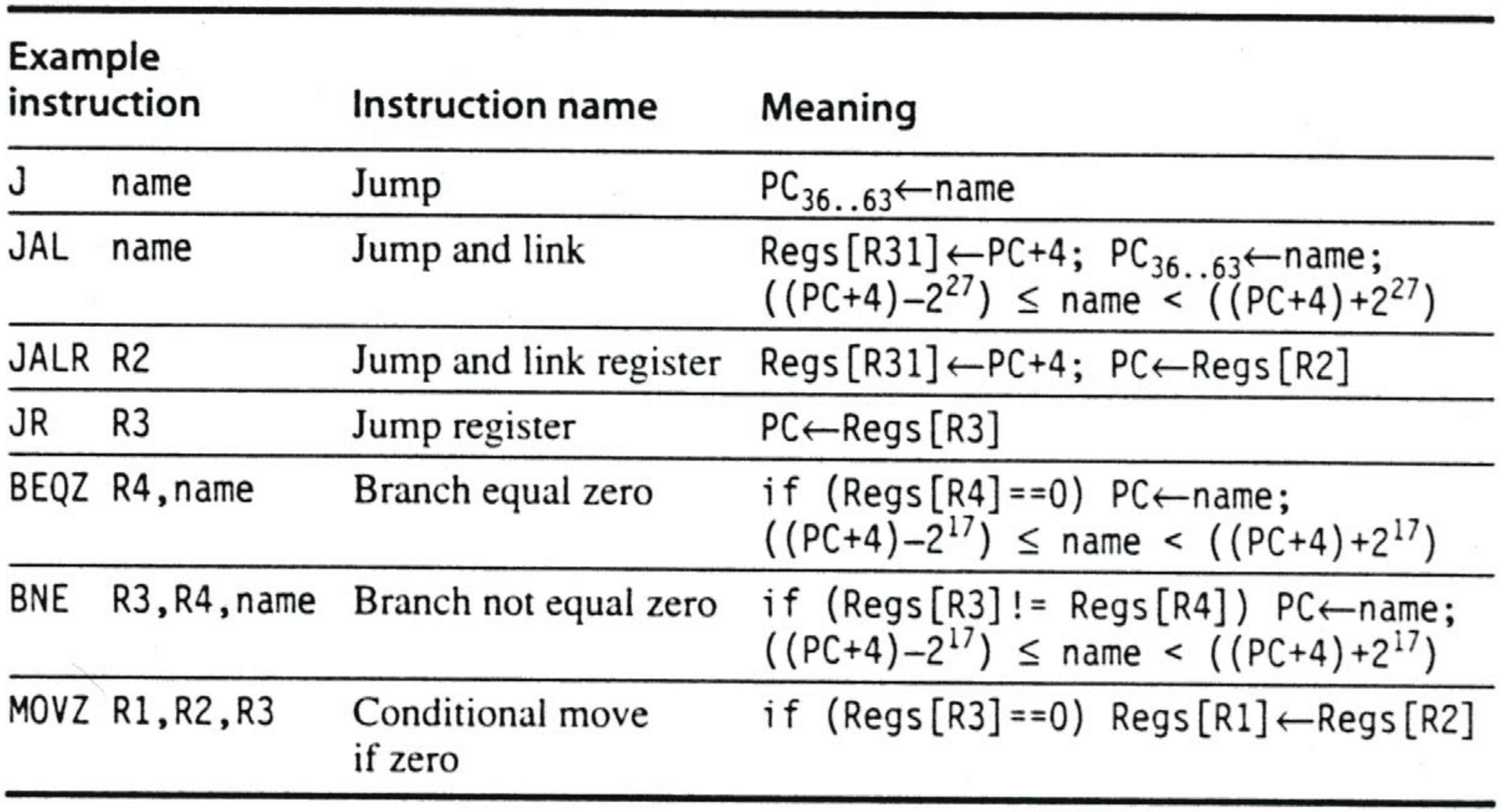
ALU Instructions



Example instruction	Instruction name	Meaning
DADDU R1,R2,R3	Add unsigned	Regs[R1]$\leftarrow$Regs[R2]+Regs[R3]
DADDIU R1,R2,#3	Add immediate unsigned	Regs[R1]$\leftarrow$Regs[R2]+3
LUI R1,#42	Load upper immediate	Regs[R1]$\leftarrow 0^{32}$##42##0^{16}
DSLL R1,R2,#5	Shift left logical	Regs[R1]$\leftarrow$Regs[R2]<<5
SLT R1,R2,R3	Set less than	if (Regs[R2]<Regs[R3]) Regs[R1]$\leftarrow$1 else Regs[R1]$\leftarrow$0

Control Flow Instructions

- Branches work in conjuction with set (e.g. SLT)



Example instruction		Instruction name	Meaning
J	name	Jump	$PC_{36..63} \leftarrow$ name
JAL	name	Jump and link	Regs[R31] $\leftarrow$ PC+4; $PC_{36..63} \leftarrow$ name; $((PC+4)-2^{27}) \le$ name $< ((PC+4)+2^{27})$
JALR	R2	Jump and link register	Regs[R31] $\leftarrow$ PC+4; PC $\leftarrow$ Regs[R2]
JR	R3	Jump register	PC $\leftarrow$ Regs[R3]
BEQZ	R4,name	Branch equal zero	if (Regs[R4]==0) PC $\leftarrow$ name; $((PC+4)-2^{17}) \le$ name $< ((PC+4)+2^{17})$
BNE	R3,R4,name	Branch not equal zero	if (Regs[R3]!= Regs[R4]) PC $\leftarrow$ name; $((PC+4)-2^{17}) \le$ name $< ((PC+4)+2^{17})$
MOVZ	R1,R2,R3	Conditional move if zero	if (Regs[R3]==0) Regs[R1] $\leftarrow$ Regs[R2]

Floating Point

ADD.D	ADD. S	ADD. PS
SUB.D	SUB.S	SUB.PS
MUL.D	MUL.S	MUL.PS
MADD.D	MADD.S	MADD.PS
DIV.D	DIV.S	DIV.PS

CVT._._

_ = L, W, D, S

C.__.D C.__.S
(__ = LT, GT, LE, GE, EQ, BE, uses FP status register)

Pitfalls and Fallacies

- **Pitfall**: *Computers with "high-level" instruction set features (Lisp, Pascal, procedure calls) have constantly failed*. Compiler/interpreters always won over hardware.

- **Fallacy**: *There are typical programs*. See the trouble in setting SPEC standards.

- **Pitfall**: *Introducing new instructions to reduce code size without accounting for the compiler*. Start with tightest compilation before proposing hardware innovations.

- **Pitfall**: *Expecting to get good performane from a compiler for DSPs.*
- There is a lot of room for hand coding in assembler.

- **Fallacy**: *An architecture with flaws cannot be successful.* Intel 80x86 made bad architectural decisions that yet has been enormously popular.

- **Fallacy**: *There are flawless designs.* Technology changes. Software/hardware trade-offs become invalid.