Topic 2

Instruction Set Architectures

(Appendix B)

ECSE 425

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

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 Appendix B of the course text.

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

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").

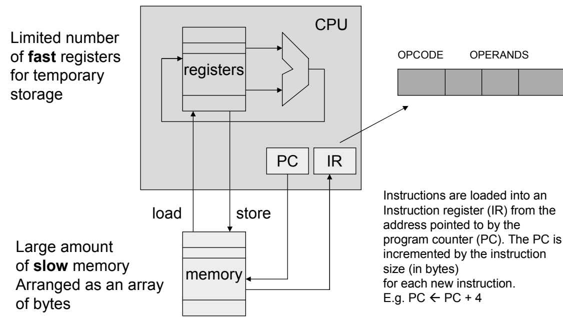
© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

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

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

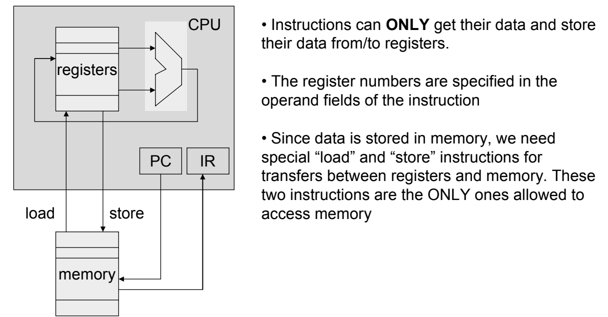
Basic Computer Organization



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

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

Load/Store Architecture (Reg-Reg)



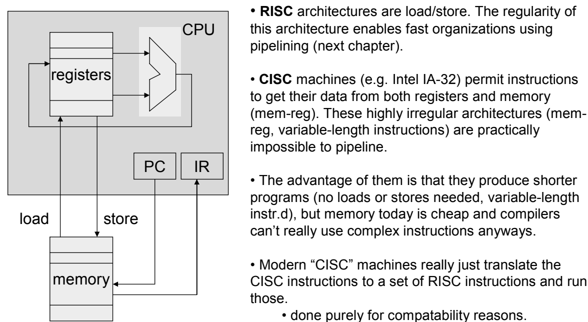
• Instructions can **ONLY** get their data and store their data from/to registers.

• 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

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

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.

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

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.

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

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
```

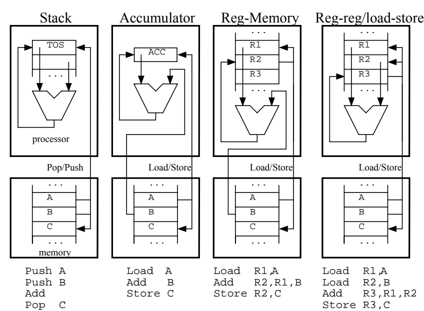
© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

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)

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

C = A + B



© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

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).

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

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

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

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)

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

```
Stack          mem.   Accumulator     mem.   Memory/Memory              mem
PUSH A          x     LOAD  A          x     MUL TMP1, A, B             xxx
PUSH B          x     MUL   B          x     MUL TMP2, C, D             xxx
MUL                   STORE TMP        x     ADD res, TMP1, TMP2        xxx
PUSH C          x     LOAD  C          x
PUSH D          x     MUL   D          x
MUL                   ADD   TMP        x
ADD                   STORE res        x
POP res         x
8 inst. /  5 mem.     7 inst. / 7 mem.       3 inst. / 9 mem.

Register-Memory      mem.      Register-Register     mem.
LOAD  R1, A           x        LOAD  R1, A            x
MUL   R1, B           x        LOAD  R2, B            x
STORE R1, TMP         x        MUL   R3, R1, R2
LOAD  R1, C           x        LOAD  R1, C            x
MUL   R1, D           x        LOAD  R2, D            x
ADD   R1, TMP         x        MUL   R4, R1, R2
STORE R1, mem         x        ADD   R5, R3, R4
                               STORE R5, mem          x
7 inst. / 7 mem.               8 inst / 5 mem
```

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

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

C

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

Byte Ordering

Big Endian Byte order puts the byte whose address is “x…x000” at the most-significant position in the double word (the big end)

Big Endian

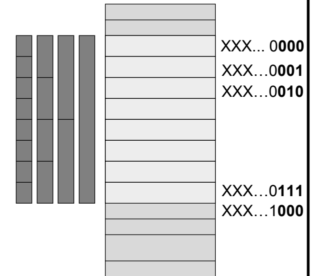
Little Endian Byte order puts the byte whose address is “x…x000” at the least-significant position in the double word (the little end)

Little Endian

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

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, …)
- An access to an object of size s bytes at byte address A is aligned if
  - A mod s=0



© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

Addressing Modes

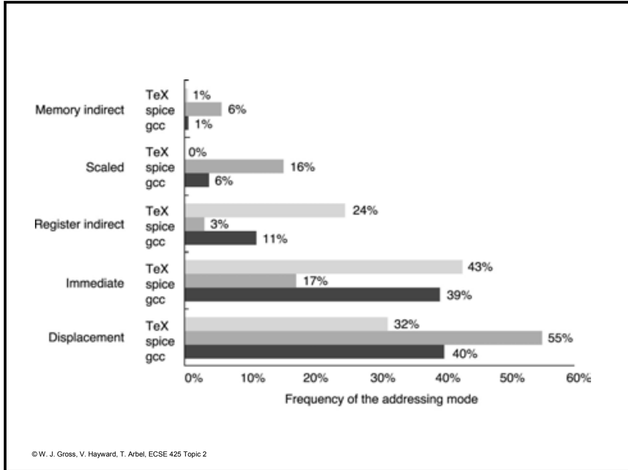
- How architectures specify the address of an object they will access

Addressing mode	Example instruction	Meaning	When used
Register	Add R4,R3	Regs[R4]←Regs[R4]+ Regs[R3]	When a value is in a register
Immediate	Add R4,#3	Regs[R4]←Regs[R4]+3	For constants
Displacement	Add R4,100(R1)	Regs[R4]←Regs[R4]+ Mem[100+Regs[R1]]	Accessing local variables
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

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

Addressing mode	Example instruction	Meaning	When used
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

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2



Summary: Memory Addressing

- Support addressing modes (popularity, 75% to 99% of the addressing modes used in measurements):
  – Displacement
    - Size of the address at least 12-16 bits (capture 75% to 99% of the displacements)
  – Immediate
    - Size of the immediate field at least 8-16 bits (50% - 8 bits, 80% - 16)
  – register indirect

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)

Operations

Operator type	Examples
Arithmetic and Logical	Integer arithmetic and logical operations: add, subtract (signed, unsigned), and, or, shifts, multiply, divide.
Data Transfer	Load, Store (move instructions on computers with memory addressing)
Control	Branch, jump, procedure call, return, traps.
System	Operating system call, Virtual memory management instructions
Floating Point	Floating Point operations: add, multiply, divide, compare
Decimal	Decimal add, decimal multiply, decimal-to-character conversions
Strings	Move, copy, compare, search
Graphics	Pixel and Vertex operations, compress/decompress operations

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 the 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	How condition is tested	Advantage	Disadvantage
Condition Code (CC)	80x86 ARM PowerPC Sparc	Tests special bits set by ALU operations, possibly under program control	Condition can be set at no cost	CC is extra state. Constrains ordering of instructions since information is passed from one instruction to a branch.
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	Compare is part of the branch. Often compare is limited to subset	One instruction rather than two for a branch	Hard to pipeline.

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

Summary: Instructions for Control Flow

- Control flow instructions – the most frequently executed
- Branch addressing
  - Jump to hundreds of instructions (above or below the branch)
    - PC-relative branch displacement of at least 8 bits
- Register indirect and PC-relative addressing for jump instructions to support returns and other features

Instruction architecture

- Seen by an assembly language programmer or compiler writer
  - Load-store architecture
    - Displacement, Immediate, Register indirect addressing modes
  - Data
    - 8-, 16-, 32-, and 64-bit integers and 32- and 64-bit floating point
  - Instructions
    - Simple operations, PC-relative conditional branches, jump and link instructions for procedure call, and register indirect jumps for procedure return

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).**
- **Size of instructions**
  - **Number of registers**
  - **Number of addressing modes**

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

Instruction Encoding

- **Architect must balance**
  - **Desire to have as many registers and addressing modes as possible**
  - **Impact of the size of the register and addressing mode fields on the average instruction size (program size)**
  - **Desire to have instructions encoded into lengths that will be easy to handle in a pipelined implementation**

Instruction Encoding

Popular Choices for encoding the instruction set

- **fixed length (Alpha, ARM, MIPS, Power PC, SPARC)**
  - **fixed number of operands**
  - **combines operation and addressing mode into opcode**
  - **Fixed instruction length, larger code representation, easy to decode.**

Operation	Address field 1	Address field 2	Address field 3

- **variable length (Intel 80x86, VAX)**
  - **any number of operands, permits all addressing modes**
  - **Flexible instruction length, smaller code representation, harder to decode.**

Operation and no. of operands	Address specifier 1	Address field 1	. . .	Address specifier n	Address field n

Instruction Encoding

Popular Choices for encoding the instruction set

- **Hybrid (IBM 360/370, MIPS16, Thumb)**
  - **Provide multiple instruction lengths**

Operation	Address specifier	Address field

Operation	Address specifier 1	Address specifier 2	Address field

Operation	Address specifier	Address field 1	Address field 2

Summary: Encoding an Instruction Set

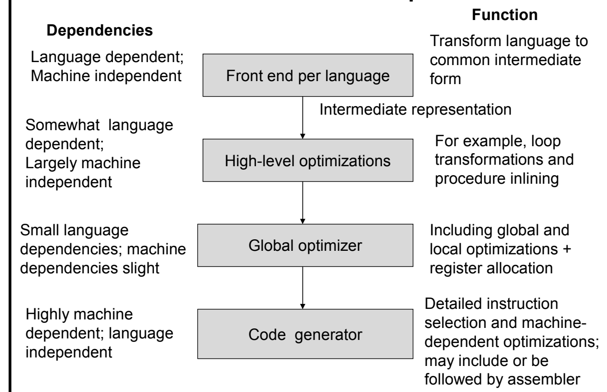
- Variable encoding
  - Optimizing code size
- Fixed encoding
  - Optimizing performance

The role of Compilers

- ISA is a compiler target
- Compiler affects the performance of a computer
  - Understanding compiler technology is critical
    - To design and efficiently implement an instruction set
- Architectural choices affect the quality of the code
  - Complexity of building a compiler
- Increased **role of compilers** in system design
  - balance the job of the hardware and that of the software
  - These are no longer separate problems

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

The role of Compilers



Compilers

- Goals:
  - All correct programs compile/work correctly
  - Most compiled programs execute quickly
  - Most programs compile quickly
  - Interoperability among languages
  - Provide debugging support

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

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)

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

Optimizations

- **High-level optimizations**
  - **Procedure integration**
- **Local (within a straight-line code fragment):**
  - **Common subexpression 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

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

Optimizations

- **Register allocation**
  - Associates registers with operands
  - Speeds up the code
  - Makes other optimizations useful
  - Based on graph coloring
    - Construct a graph representing the possible candidates for allocation to a register
    - Use a limited set of colors so that no two adjacent nodes have the same color
- **Processor-dependent optimizations**
  - attempt to take advantage of specific architectural knowledge
    - Strength reduction, Pipeline scheduling …

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

Example

- **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!

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

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.
  - Simplify trade-offs among alternatives
  - Provide instructions that bind the quantities known at compile time as constants

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

Summary

- We expect a new ISA
  - At least 16 GPR (+ FPR)
  - All supported addressing modes apply to all instructions that transfer data
  - Provide primitives
  - Simplify trade-offs between alternatives
  - Don't bind constants at run time
- => simplicity!

MIPS-64 ISA

- Will be used for rest of the course
- A classic RISC ISA
- MIPS emphasizes
  - Simple load-store instruction set
  - Design for pipelining efficiency, including a fixed instruction set encoding
  - Efficiency as a compiler target
- Subset of what is now called MIPS64

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

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

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

Data Types

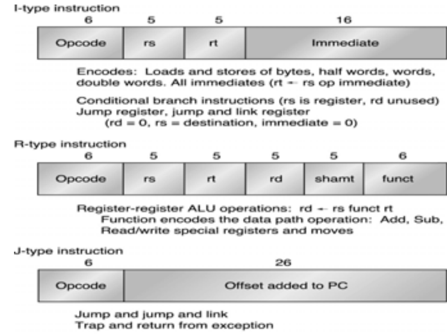
- 8-bit bytes
- 16-bit half-words
- 32-bit words
- 64-bit double words
- 32-bit single-precision FP
- 64-bit double-precision FP

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

Addressing

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

Instruction Format



© 2003 Elsevier Science (USA). All rights reserved.

Operations

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

- More on RTL.
  - $\leftarrow_n$ transfer *n* bits, *x,y* ← *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, 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)

Regs[R10]$_{32\ldots63}$ $\leftarrow_{32}$ (Mem[Regs[R8]]$_0$)^{24} ## 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]←Regs[R2]+Regs[R3]
DADDIU R1,R2,#3	Add immediate unsigned	Regs[R1]←Regs[R2]+3
LUI R1,#42	Load upper immediate	Regs[R1]←0^{32}##42##0^{16}
DSLL R1,R2,#5	Shift left logical	Regs[R1]←Regs[R2]<<5
DSLT R1,R2,R3	Set less than	if (Regs[R2]<Regs[R3]) Regs[R1]←1 else Regs[R1]←0

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

Control Flow Instructions

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

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

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

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)

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2

Pitfalls and Fallacies

- **Pitfall**: *Designing a “high-level” instruction set feature specifically oriented to supporting a high-level language structure.*
- **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.
- **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.

© W. J. Gross, V. Hayward, T. Arbel, ECSE 425 Topic 2