

Computer Organization

Instruction Set Architecture

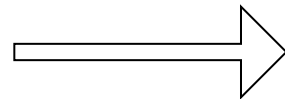
B.Tech. II (CSE)

Instruction Set Architecture

■ C code

A=b+c;

D=e+f;

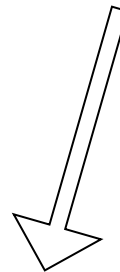


Compiler

■ Assembly Code

Add \$s3, \$s2, \$s1

Add \$s7, \$s5, \$s6



Encoding straight
forward

■ Machine Code

...0...1..

...0...1..

Instruction Set Architecture

■ C code

A=b+c;

D=e+f;

Machine
Independent

■ Assembly Code

Add \$s3, \$s2, \$s1

Add \$s7, \$s5, \$s6

Compiler

Encoding straight
forward

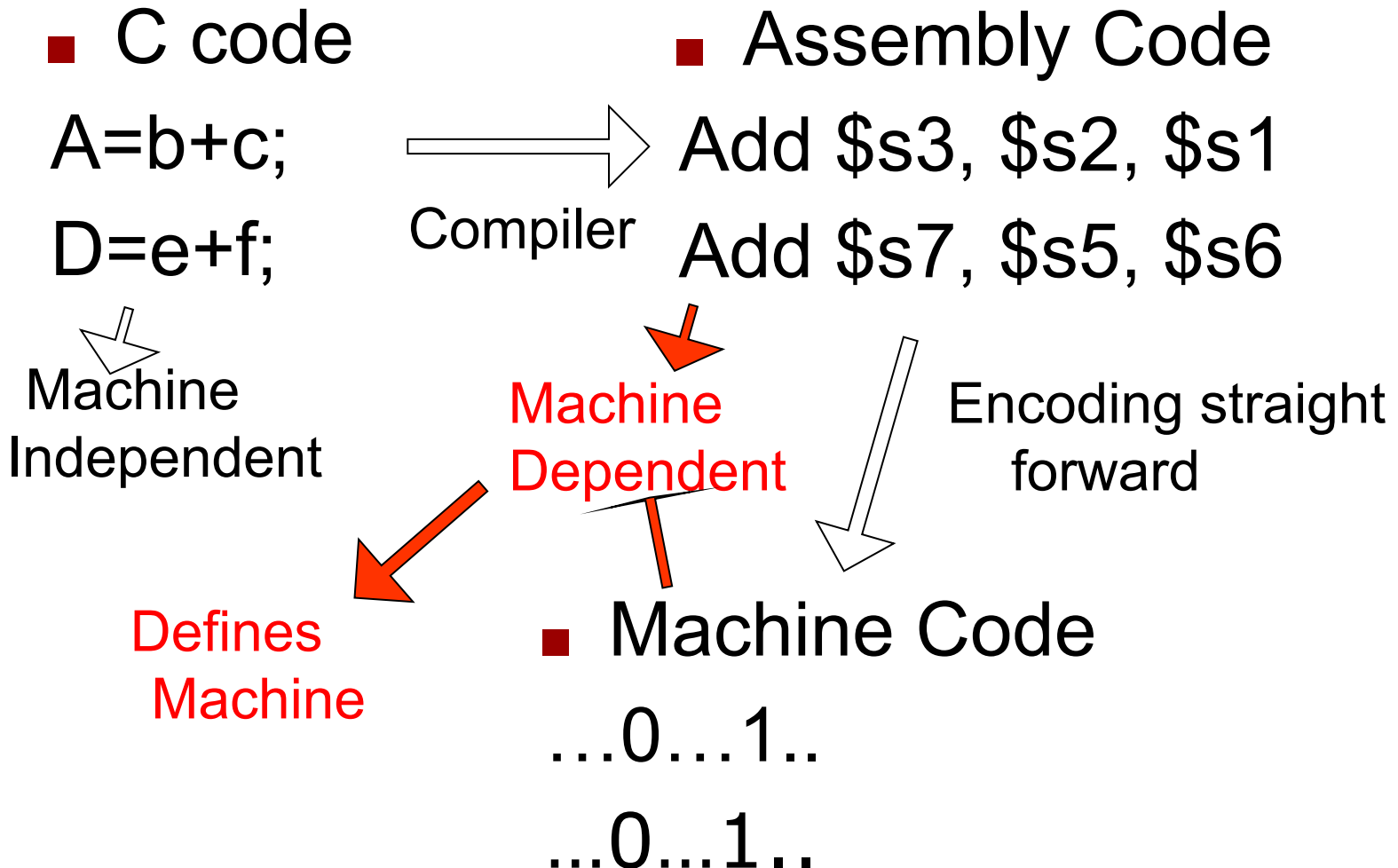
Machine
Dependent

Defines
Machine

■ Machine Code

...0...1..

...0...1..



Instruction Set Architecture

■ C code

A=b+c;

D=e+f;



MIPS

The diagram illustrates the translation of C code to different instruction set architectures. It features a central C code block with two lines: 'A=b+c;' and 'D=e+f;'. Three arrows originate from this block: one pointing to 'MIPS' on the left, one pointing to 'X86' in the center, and one pointing to 'ARM' on the right. The arrows are double-lined with open arrowheads. The text 'MIPS', 'X86', and 'ARM' are in a large, bold, black sans-serif font.

X86

ARM

Instruction Set Architecture

■ C code

A=b+c;

D=e+f;

MIPS
Instruction
Set

MIPS

Assembly
code

X86
Instruction
Set

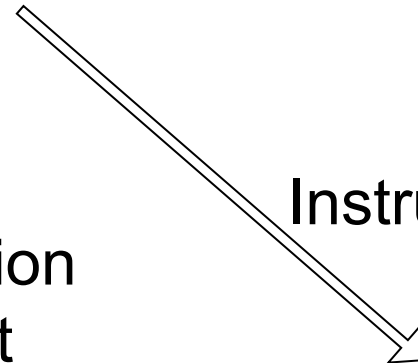
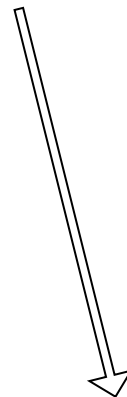
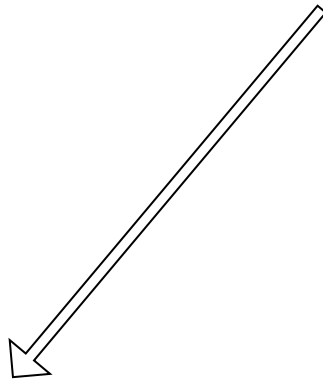
X86

Assembly
Code

ARM
Instruction Set

ARM

Assembly
Code



Instruction Set Architecture

■ Assembly Code

Add \$s3, \$s2, \$s1

Add \$s7, \$s5, \$s6

Instruction set is the interface
between hardware and
software

Instruction Set Design

- Central part of any system design
- Allows abstraction, independence

Why?

- Early days, new computer having its own new set of instructions
- Needed to allow backward compatibility

Topics

- Instruction Set Architecture
- Key of ISA using MIPS
 - ✱ Design Principles
 - ✱ Instructions
 - ✱ Instruction formats
 - ✱ Addressing modes

ISA

ISA or Instruction Set

- The level - between the high-level languages and the hardware
- When new hardware architecture comes along ...
 - ✱ Can add new features to exploit new hardware capabilities
 - ✱ Need to maintain backward compatibility

ISA

ISA-level code is what a compiler outputs

ISA

- ISA-level code is what a compiler outputs
- Compiler writer needs to know
 - ✱ Memory model
 - ✱ Types of registers are available
 - ✱ What instructions are available
 - Instruction formats
 - Opcodes
 - ✱ Exceptional conditions

ISA

- An ISA includes a specification of the set of opcodes (machine language), the native commands implemented by a particular processor
- Related to programming includes
 - ✱ Native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O

ISA

- Distinguished from the microarchitecture
 - ✱ MAL which is the set of processor design techniques used to implement the instruction set
- Computers with different microarchitectures can share a common instruction set
- For example:
 - ✱ The [Intel](#) The Intel [Pentium](#) The Intel Pentium and the [AMD](#) The Intel Pentium and the AMD [Athlon](#) The Intel Pentium and the AMD Athlon implement nearly identical versions of the [x86](#) instruction set, but have radically different internal designs

ISA

■ Stored Program Concept

✱ Fetch & Execute Cycle

- Instructions are ***fetched*** and put into a special register
- Bits in the register ***control*** the *subsequent actions* (= *execution*)
- Fetch the next instruction and ***repeat***

■ Instructions

- ✱ Encoded in binary, called machine code

ISA Instructions

- More primitive than higher level languages,
 - ✱ e.g., no sophisticated control flow such as *while* or *for* loops
- Different computers have different instruction sets
 - ✱ But with many aspects in common
- Computers have very simple instruction sets
 - ✱ Makes the Implementation Simple

Instruction Set

- The complete collection of instructions that are understood by a CPU
 - ✱ Can be considered as a functional spec for a CPU
 - Implementing the CPU in large part is implementing the machine instruction set
- Machine Code is rarely used by humans
 - ✱ Binary numbers / bits
 - ✱ Usually represented by human readable assembly codes
 - ✱ In general, one assembler instruction equals one machine instruction

Elements of an Instruction

- Operation code (Op code)
 - ✱ Do this
- Source Operand reference
 - ✱ To this
- Result Operand reference
 - ✱ Put the result here
- Next Instruction Reference
 - ✱ When you have done that, do this...
 - ✱ Next instruction reference often implicit (sequential execution)

Operands

- Main memory (or virtual memory or cache)
 - ✱ Requires address
- CPU register
- I/O device
 - ✱ Several forms:
 - Specify I/O module and device
 - Specify address in I/O space
 - Memory-mapped I/O just another memory address

Sample Instruction Format



Key of ISA

Operations

- What operations are provided??

Operands

- How many? how big?
- How are memory addresses computed?

How many registers?

Where do operands reside?

- e.g., can you add contents of memory to a register?

Instruction length

- Are all instructions of the same length?

Instruction format

- Which bits designate for what purpose??

Operations OR Instruction Types

- Data processing
 - ✱ Arithmetic and logical instructions
- Data storage (main memory)
- Data movement (I/O)
- Program flow control
 - ✱ Conditional and unconditional branches
 - ✱ Call and Return

ISA Architecture Types

Classification according to,

- Type of INTERNAL STORAGE in CPU
- Type and no. of OPERANDS

ISA Architecture Types

- In the CPU, type of INTERNAL STORAGE is the most basic differentiation in ISA
 - ✱ Stack, Accumulator or Set of registers
- Accordingly architectures are named:
 - ✱ Stack architecture
 - ✱ Accumulator architecture
 - ✱ Register architecture

ISA Architecture Types

- Operands may be named explicitly or implicitly
 - ✱ Stack architecture
 - Implicitly on the top of the stack
 - ✱ Accumulator architecture
 - One operand is implicitly the accumulator
 - ✱ General-purpose register architectures
 - Only explicit operands—either registers or memory locations
 - Operands may be accessed directly from memory or may need to be first loaded into temporary storage, depending on the class of instruction and choice of specific instruction

ISA

- Classification of Register Architecture according to the type of operands
 - ✱ Load-store or register-register machines
 - With no memory reference per ALU instruction
 - ✱ Register-memory
 - Instructions with one memory operands per typical ALU instruction
 - ✱ Memory-memory
 - Instructions with one or more than one memory operand

ISA ISA Architecture Types

- Code $C=A+B$,
- On these three classes of instruction sets where A , B and C all belong in Memory

1. Stack	2. Accumulator	3. Register	
		Register-Memory	Load-Store

ISA ISA Architecture Types

- Code $C=A+B$,
- On these three classes of instruction sets where A, B and C all belong in Memory

1. Stack	2. Accumulator	3. Register	
		Register-Memory	Load-Store
Push A Push B Add Pop C			

ISA ISA Architecture Types

- Code $C=A+B$,
- On these three classes of instruction sets where A, B and C all belong in Memory

1. Stack	2. Accumulator	3. Register	
		Register-Memory	Load-Store
Push A Push B Add Pop C	Load A Add B Store C		

ISA

Classes of register architecture

3.1 Register-memory architecture

Can access memory as part of any instruction

3.2 Load-store or register-register architecture

3.3 Memory-memory architecture

1. Stack	2. Accumulator	3. Register	
		Register-Memory	Load-Store
Push A Push B Add Pop C	Load A Add B Store C	Load R1, A Add R1, B Store C, R1	

ISA

Classes of register architecture

3.1 Register-memory architecture

Can access memory as part of any instruction

3.2 Load-store or register-register architecture

Can access memory only with load and store instructions

3.3 Memory-memory architecture

1. Stack	2. Accumulator	3. Register	
		Register-Memory	Load-Store
Push A Push B Add Pop C	Load A Add B Store C	Load R1, A Add R1, B Store C, R1	Load R1, A Load R2, B Add R3, R1, R2 Store C, R3

ISA

- Third class of register architecture

3.3 Memory-Memory architecture

- Keeps all operands in memory
- **Not found in today's machines**

1. Stack	2. Accumulator	3. Register	
		Register-Memory	Load-Store
Push A Push B Add Pop C	Load A Add B Store C	Load R1, A Add R1, B Store C, R1	Load R1, A Load R2, B Add R3, R1, R2 Store C, R3

ISA

General Two classes of Register Architecture

3.1 Register-memory architecture

- Can access memory as part of any instruction

3.2 Load-store or register-register architecture

- Can access memory only with load and store instructions

1. Stack	2. Accumulator	3. Register	
		Register-Memory	Load-Store
Push A Push B Add Pop C	Load A Add B Store C	Load R1, A Add R1, B Store C, R1	Load R1, A Load R2, B Add R3, R1, R2 Store C, R3

Utilized in today's machine

ISA

- Example Code $(A*B)-(C*D)-(E*F)$
- On a stack architecture
 - ✱ Must be evaluated left to right, unless special operations or swaps of stack positions are done
 - ✱ A stack cannot be accessed randomly
- On an accumulator architecture
 - ✱ Creating lots of bus traffic
- On a register architecture
 - ✱ May be evaluated by multiplying in any order, which may be **more efficient** because of the location of the operands or because of pipelining

ISA

- Most Early Machines used
 - ✱ Stack or Accumulator-style architectures
 - ✱ Dedicating components / registers for special uses
 - Less number of general-purpose registers
 - Trying to allocate variables to registers will not be profitable

ISA-Load-Store Reg. Architecture

- Machines designed after 1980 uses a load-store register arch., the registers are used for variables
 - ✱ To reduce memory traffic
 - ✱ To speed up the program
 - As registers are **faster** than memory
 - ✱ To improve the code density
 - **Fewer bits** are needed to represent the register than the memory location
- Registers are **easier for a compiler to use and can be used more effectively** than other forms of internal storage

ISA-Load-Store Reg. Architecture

- How many registers are sufficient?
 - ✱ Answer depends on how they are used by the compiler
- Most compilers reserve
 - Some registers for expression evaluation
 - Some for parameter passing
 - Remainder to be allocated to hold variables

ISA

- GPR's major concern-the no. of operands for a typical arithmetic or logical instruction
 1. Whether **ALU instruction has two or three operands**
 - 3-operand instruction format
 - Instruction contains a result and two source operands
 - 2-operand instruction format
 - One of the operands is both a source and a result for the operation
 2. **How many of the operands may be memory addresses in ALU instructions**
 - May vary from none to three

ISA

- Summary of Classification of Architectures according to the type of operands

ISA – GPR Architecture

1) Register-register (0-Memory + 3-Reg = Total 3)

✱ Advantage

- Simple, fixed-length instruction encoding
- Simple code-generation model
- Instructions take similar numbers of clocks to execute

✱ Disadvantage

- Higher instruction count than architectures having memory references in instructions
- Some instructions are short and bit encoding may be wasteful

✱ Example - SPARC, MIPS, PowerPC, ALPHA

ISA – GPR Architecture

2) Register – memory (1- Memory + 1-Reg= Total 2)

- ✱ Advantage

- Data can be accessed without loading first
- Instruction format tends to be easy to encode and yields good density

- ✱ Disadvantage

- Operands are not equivalent since a source operand in a binary operation is destroyed
- Encoding a register number and a memory address in each instruction may restrict the number of registers
- Clocks per instruction varies by operand location

- ✱ Example - Intel 80x86, Motorola 68000

ISA – GPR Architecture

3) Memory-memory (3-Memory + 0-Reg = Total-3)

✱ Advantage

- Most compact
- Doesn't waste registers for temporaries

✱ Disadvantage

- Large variation in instruction size, especially for three-operand instructions
- Also, large variation in work per instruction
- Memory accesses create memory bottleneck

✱ Example - VAX

ISA

- Summary, In general,
 - ✱ Machines with **fewer alternatives** make the **compiler's task simpler** since there are fewer decisions for the compiler to make
 - ✱ Machines with a **wide variety** of flexible instruction formats **reduce the number of bits** required to encode the program
 - ✱ A machine that uses a small number of bits to encode the program is said to have good instruction density—a smaller number of bits do as much work as a larger number on a different architecture
 - ✱ The no. of registers also affects the instruction size

Operands

- How many operands are supported?
 - ✱ 3 operands
 - ✱ 2 operands
 - ✱ 1 operand
 - ✱ 0 operand

No. of Operands

- 3 operands
 - ✱ Operand 1, Operand 2, Result
 - ✱ $a = b + c;$
 - ✱ `add ax, bx, cx`
 - ✱ May be a fourth address - next instruction (usually implicit)[not common]
- Instructions are long because 3 or more operands have to be specified

No. of Operands

■ 2 Operands

- ✱ One address doubles as operand and result
- ✱ $a = a + b$
- ✱ `add ax, bx`
- ✱ Reduces length of instruction over 3-address format
- ✱ Requires some extra work by processor
- ✱ Temporary storage to hold some results

No. of Operands

- 1 Operand
 - ✱ Implicit second address
 - ✱ Usually a register (accumulator)
 - ✱ Common on early machines
- Used in some Intel x86 instructions with implied operands
 - ✱ `mul ax`
 - ✱ `idiv ebx`

No. of Operands

- 0 (zero) Operand
 - ✱ All addresses implicit
 - ✱ Uses a stack- X87 example $c = a + b$:
 - push a
 - push b
 - fadd //a+b, pop stack
 - store and pop c
- Can reduce to 3 instructions:
 - ✱ push a
 - ✱ push b
 - ✱ faddp c ; //add and pop

Computation of $Y = (a-b) / (c + (d * e))$

- Three Operands instructions
- Two Operands instructions
- One Operand instructions

Computation of $Y = (a-b) / (c + (d * e))$

- Three Operands instructions

- ✱ `sub y,a,b`
- ✱ `mul t,d,e`
- ✱ `add t,t,c`
- ✱ `div y,y,t`

- Two Operands instructions

- ✱ `mov y,a`
- ✱ `sub y,b`
- ✱ `mov t,d`
- ✱ `mul t,e`
- ✱ `add t,c`
- ✱ `div y,t`

Computation of $Y = (a-b) / (c + (d * e))$

- One Operand instructions

- ✱ load d
- ✱ mul e
- ✱ add c
- ✱ store y
- ✱ load a
- ✱ sub b
- ✱ div y
- ✱ store y

How Many Operands?

■ More Operands

- ✱ More complex instructions
- ✱ More registers
 - Inter-register operations are quicker
- ✱ Fewer instructions per program
- ✱ More complexity in processor

■ Fewer Operands

- ✱ Less complex instructions
- ✱ One address format however limits you to one register
- ✱ More instructions per program
- ✱ Less complexity in processor
 - Faster fetch/execution of instructions

Memory Organization

- Viewed as a large single-dimension array with access by *address*
- A memory address is an *index* into the memory array
- Two views of Memory
 - * Byte Addressing
 - The index points to a byte of memory, and that the unit of memory accessed by a load/store is a byte
 - * Word Addressing

0	8 bits of
1	data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
6	8 bits of data

...

Memory Organization

- How many bytes (8 bits) and words (32 bits) can be accessed for 4 GB Memory?
 - ✱ 2^{32} bytes with byte addresses from 0 to $2^{32}-1$
 - ✱ 2^{30} words with byte addresses 0, 4, 8, ... $2^{32}-4$
 - Words are *aligned*

Memory Organization

- Why Word alignment ?
 - ✱ Memories operate more efficiently this way
- Consider 8-byte (64-bit) words

Memory Organization

- Bytes in a word can be numbered in two ways:
 - ✱ Big Endian
 - Most-significant byte at least address of a word
 - MIPS is Big Endian
 - ✱ Little Endian
 - Least-significant byte at least address
 - ? Is Little Endian

Memory Organization

- Example: Store the number 12 in 32 bits

There will be 28 zeroes and then 1100

(MSB) 00000000 00000000 00000000 00001100 (LSB)

	Big-endian	Little-endian
Byte 0:	0000 0000	0000 1100
1:	0000 0000	0000 0000
2:	0000 0000	0000 0000
3:	0000 1100	0000 0000

The big-endian system 1100 is in **byte 3**

The little-endian system 1100 is in **byte 0**

ISA

- Example ISA's:
 - ✱ Digital's VAX (1977)
 - ✱ Intel's x86 (1978), but successful (IBM PC)
 - ✱ MIPS – focus of text, used in assorted machines
 - ✱ PowerPC – used in Mac's, IBM supercomputers, ...
- VAX and x86 are CISC (“Complex Instruction Set Computers”)
 - ✱ Started in 70's
- MIPS and PowerPC are RISC (“Reduced Instruction Set Computers”)
 - ✱ Almost all machines of 80's and 90's are RISC
 - Including VAX's successor, the DEC Alpha

RISC vs. CISC

RISC

- Instructions in Instruction set of processor are simple and few in number
- Instructions to access memory
 - only **LOAD/STORE**
- Instruction length - Fixed
- Addressing modes - Few
- Complexity in compiler
- Achieves shorten execution time by reducing the *clock cycles per instruction* (i.e. simple instructions take less time to interpret)

CISC

- Many complex instructions
- Instructions to access memory - many instructions can access
- Instruction length - Variable
- Addressing modes - Many
- Complexity in microcode
- Achieves shorten execution time by reducing the number of instructions per program

And many more as discussed in class...

Example for RISC vs. CISC

Multiplication:

CISC: Mov ax,10
 Mov bx,5
 Mul bx, ax

RISC: Mov ax,0
 Mov bx,10
 Mov cx, 5
 Begin: **Add ax,bx**
 Loop begin

- The total clock cycles for the CISC version might be:
 $(2 \text{ movs} \times 1 \text{ cycle}) + (1 \text{ mul} \times 30 \text{ cycles})$
= 32 cycles
- While the clock cycles for RISC version is:
 $(3 \text{ movs} \times 1 \text{ cycle}) + (5 \text{ adds} \times 1 \text{ cycle}) + (5 \text{ loops} \times 1 \text{ cycle})$
= 13 cycles

ISA

Design goals:

- *Maximize performance*
- *Minimize cost*
- *Reduce design time*

The MIPS

Microprocessor without Interlocked Pipeline Stages

- RISC instruction set architecture (ISA)
- Large share of embedded core market
 - ✱ Applications in consumer electronics, network / storage equipment, cameras, printers, ...
- Typical of many modern ISAs

MIPS Instruction Set

- What should be considered?
 - ✱ Operations (MIPS Arithmetic)
 - ✱ MIPS Operand
 - Register
 - Memory

Operations (MIPS Arithmetic)

Example:

- C code: $A = B + C$
- C code: $A = B + C + D + E$
- C code: $F = (G + H) - (I + J)$
- C code: $G = H + A[8];$

Operations (MIPS Arithmetic)

Example:

- C code: $A = B + C$
- All MIPS arithmetic instructions have 3 operands
- Operand order is fixed (e.g., destination first)
- MIPS code: Add A, B, C

MIPS Arithmetic

Example:

- C code: $A = B + C + D + E$

- MIPS code:

 - Add A, B, C

 - Add A, A, D

 - Add A, A, E

MIPS Arithmetic

Example:

- C code: $F = (G + H) - (I + J)$

- MIPS code:

 - Add F, G, H

 - Sub F, I, J

MIPS Arithmetic

Example:

- C code: $F = (G + H) - (I + J)$
- MIPS code: //Use of temporary variables
 - Add \$t0, G, H
 - Add \$t1, I, J
 - Sub F, \$t0, \$t1

MIPS Arithmetic

Design Principle 1:

- Simplicity favors regularity.
 - ✱ i.e. Regularity makes implementation simpler
- Simplicity enables higher performance at lower cost

MIPS Operand

- Arithmetic instructions use register operands
- MIPS has a 32×32 -bit register file
 - ✱ Use for frequently accessed data
 - ✱ 32-bit data called a “word”

MIPS Registers and Memory

MIPS Operand

- Arithmetic instructions use register operands
- MIPS has a 32×32 -bit register file
 - ✱ Use for frequently accessed data
 - ✱ 32-bit data called a “word”
- Assembler names
 - ✱ \$t0, \$t1, ..., \$t9 for temporary values
 - ✱ \$s0, \$s1, ..., \$s7 for saved variables // C variables

MIPS Operand

- Only 32 Registers?

Design Principle 2:

- Smaller is faster.
- *Why?*
 - ✱ Electronic signals have to travel further on a physically larger chip increasing clock cycle time
 - ✱ Smaller is also cheaper!

MIPS Register Operand

- C code:

$$f = (g + h) - (i + j);$$

- Compiled MIPS code:

- ✱ f, \dots, j in $\$s0, \dots, \$s4$

- Add $\$t0, \$s1, \$s2$

- Add $\$t1, \$s3, \$s4$

- Sub $\$s0, \$t0, \$t1$

MIPS Register Operand

- Arithmetic instructions operands must be in registers
 - ✱ MIPS has only 32 registers
- Compiler associates variables with registers
- What about programs with lots of variables (arrays, etc.)?

MIPS Memory Operands

- Main memory used for composite data
 - ✱ Arrays, structures, dynamic data
- To apply arithmetic operations
 - ✱ Load values from memory into registers
 - ✱ Store result from register to memory

MIPS Registers and Memory

MIPS Memory Organization

- ✱ Memory is byte addressed
- ✱ Each address identifies an 8-bit byte
- ✱ A word is 32 bits or 4 bytes
- ✱ Address must be a multiple of 4
- ✱ Words are aligned in memory
- ✱ Follows Big-Endian Ordering

MIPS Load/Store Instructions

<u>Instruction</u>	<u>Meaning</u>
Add \$s1, \$s2, \$s3	$\$s1 = \$s2 + \$s3$
Sub \$s1, \$s2, \$s3	$\$s1 = \$s2 - \$s3$
Lw \$s1, 100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$
Sw \$s1, 100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$

Instruction Format: R Type

MIPS Operand - Register

Register 1, called \$at, is reserved for the assembler; registers 26-27, called \$k0 and \$k1 are reserved for the operating system

- ✱ Require 5 bits to select one register

Instruction Format: R Type

Opcode and Operand

3 Registers Operands

15 bits for Register Operands

Opcode

Instruction Format: R Type

op	rs	rt	rd	shamt	funct		
----	----	----	----	-------	-------	--	--

opcode – first **second** **register** **shift** **function field -**
operation **register** **register** **destin-** **amount** **selects variant**
source **source** **ation** **00000 for of operation**
operand **operand** **operand** **now** **extends opcode**

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
--------	--------	--------	--------	--------	--------

R Type Format Example

op	rs	rt	rd	shamt	funct		
opcode – operation	first register source operand	second register source operand	register destin- ation operand	shift amount 00000 now	function field - selects variant of operation extends opcode		
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits		

Add \$t0, \$s1, \$s2

\$t0 – \$t7: Registers are: 8 – 15
 \$t8 – \$t9: Registers are: 24 – 25
 \$s0 – \$s7: Registers are: 16 – 23

special	\$s1	\$s2	\$t0	0	add
---------	------	------	------	---	-----

0	17	18	8	0	32
---	----	----	---	---	----

000000	10001	10010	01000	00000	100000
--------	-------	-------	-------	-------	--------

$$00000010 \ 00110010 \ 01000000 \ 00100000_2 = 02 \ 32 \ 40 \ 20_{16}$$

R Type Format Example

Sub \$t0, \$s1, \$s2

\$t0 – \$t7: Registers are: 8 – 15

\$t8 – \$t9: Registers are: 24 – 25

\$s0 – \$s7: Registers are: 16 – 23

special	\$s1	\$s2	\$t0	0	sub
---------	------	------	------	---	------------

0	17	18	8	0	34
---	----	----	---	---	-----------

000000	10001	10010	01000	00000	100010
--------	-------	-------	-------	-------	--------

00000010 00110010 01000000 00100000₂ = **02 32 40 22**₁₆

MIPS Instructions

Design Principle 3:

- Good design demands good compromises
 - ✱ Different formats complicate decoding
 - ✱ Keep formats as similar as possible

Immediate Operands

- Small constants are used quite frequently (50% of operands)
- Make operand part of instruction itself!
- Introduce a new type of instruction format with Immediate operands
- Design Principle 4: *Make the common case fast*
- *Example*:
 - `addi $sp, $sp, 4` # $\$sp = \$sp + 4$, $\$sp=29$
 - `addi $t0, $t0, -5` # $\$t0 = \$t0 - 5$

Immediate Operands

- For example:
 - ✱ Constant data specified in an instruction
 - ✱ `Addi $s3, $s3, 4` # $\$s3 = \$s3 + 4$
 - ✱ `Addi $sp, $sp, 4` # $\$sp = \$sp + 4$
- No subtract immediate instruction
 - ✱ Just use a negative constant
 - ✱ `Addi $s2, $s1, -1`

Instruction Format: I Type

op	rs	rt	constant	or address
6 bits	5 bits	5 bits	16 bits	
opcode – operation	first register source operand	second register source operand	constant: -2^{15} to $+2^{15} - 1$ address: offset added to base address in rs	

Immediate Operands

op	rs	rt	16 bit number
6 bits	5 bits	5 bits	16 bits

- *Example:* `addi $sp, $sp, 4` # $\$sp = \$sp + 4$, $\$sp=29$

001000	11101	11101	00000000000000100
6 bits	5 bits	5 bits	16 bits

Instruction Format

■ Load Instruction

✱ Lw \$s1, 100(\$s2)

- Two Registers
- A Constant
 - If consider, the third register to store this
 - Would be limited to 5 bits only i.e. upto 32
 - This may be larger than 32
 - So, 5-bit field is too small

?

MIPS Load/Store Instructions

- Load word has destination first
- Store has destination last
- MIPS arithmetic operands are registers, not memory locations
 - ✱ Therefore, words must first be moved from memory to registers using loads before they can be operated on; then result can be stored back to memory

MIPS Load/Store Instructions

- C code:

$G = H + A[8];$

G in \$s1, H in \$s2, base address of A in \$s3

- Compiled MIPS code:

Index 8 requires offset of 32, due to 4 bytes/word

Value Offset Base address
Lw \$t0, 32(\$s3) # load word
Add \$s1, \$s2, \$t0

MIPS Load/Store Instructions

- C code:

$A[12] = H + A[8];$

- MIPS code:

?

Load : Lw \$t0, 32(\$s3)

Arithmetic : Add \$t0, \$s2, \$t0

Store : Sw \$t0, 48(\$s3)

Instruction Format: I Type

op	rs	rt	constant	offset	address
----	----	----	----------	--------	---------

6 bits	5 bits	5 bits	16 bits
--------	--------	--------	---------

opcode – first second
operation register register
source source
operand operand

constant: -2^{15} to $+2^{15} - 1$
**address: offset added to
base address in rs**

lw \$t0, 1002(\$s2)

\$t0 – \$t7: Registers are: 8 – 15

\$t8 – \$t9: Registers are: 24 – 25

\$s0 – \$s7: Registers are: 16 – 23

100011	10010	01000	0000001111101010
--------	-------	-------	------------------

Example : I Type Format

- C code:

$A[300] = H + A[300];$

- MIPS code:

- ✱ Lw \$t0, 1200(\$t1)

- ✱ Add \$t0, \$s2, \$t0

- ✱ Sw \$t0, 1200(\$t1)

Example: I Type Format

op	rs	rt	rd	shamt/ address	funct	
35	9	8		1200		
0	18		8	8	0	32
43	9	8		1200		

\$t0 – \$t7: Registers are: 8 – 15

\$t8 – \$t9: Registers are: 24 – 25

\$s0 – \$s7: Registers are: 16 – 23

Lw \$t0, 1200(\$t1)

Add \$t0, \$s2, \$t0

Sw \$t0, 1200(\$t1)

Logical Operations

- Instructions for bitwise manipulation

Operation	C	MIPS
Shift Left	<<	sll
Shift Right	>>	srl
Bitwise AND	&	and, andi
Bitwise OR		or, ori
Bitwise NOT	~	nor

- Useful for extracting and inserting groups of bits in a word

Shift Operations

op	rs	rt	rd	shamt	funct		
6 bits		5 bits		5 bits	5 bits	5 bits	6 bits

- shamt: how many positions to shift
- Shift left logical
 - ✱ Shift left and fill with 0 bits
 - ✱ Sll by i bits multiplies by 2^i
- Shift right logical
 - ✱ Shift right and fill with 0 bits
 - ✱ Srl by i bits divides by 2^i (unsigned only)

AND Operations

- Useful to mask bits in a word
 - ✱ Select some bits, clear others to 0
- Example:

And \$t0, \$t1, \$t2 # \$t0 = \$t1 & \$t2

\$t2 = 0000 0000 0000 0000 0000 1101 1100 0000

\$t1 = 0000 0000 0000 0000 0011 1100 0000 0000

\$t0 = 0000 0000 0000 0000 0000 1100 0000 0000

OR Operations

- Useful to include bits in a word
 - ✱ Select some bits to 1, leave others unchanged
- Example:

Or \$t0, \$t1, \$t2 # \$t0 = \$t1 | \$t2

\$t2 = 0000 0000 0000 0000 0000 1101 1100 0000

\$t1 = 0000 0000 0000 0000 0011 1100 0000 0000

\$t0 = 0000 0000 0000 0000 0011 1101 1100 0000

The Constant Zero

- MIPS register 0 (\$zero) is the constant 0
 - ✱ Cannot be overwritten
- Useful for common operations
 - ✱ E.g., move between registers:
 - Value of \$s1 to \$t2
 - Add \$t2, \$s1, \$zero

NOT Operations

- Useful to invert bits in a word
 - ✱ Change 0 to 1, and 1 to 0
 - ✱ MIPS has NOR 3-operand instruction
 - ✱ $a \text{ NOR } b == \text{NOT} (a \text{ OR } b)$

- Example:

Nor \$t0, \$t1, \$zero # Register 0: always read as zero

\$t1 = 0000 0000 0000 0000 0011 1100 0000 0000

\$t0 = 1111 1111 1111 1111 1100 0011 1111 1111

Conditional Operations

- Decision making instructions
 - ✱ alter the control flow,
 - i.e., change the next instruction to be executed

MIPS conditional instructions

- Branch to a labeled instruction if a condition is true
- Otherwise, continue sequentially
 - ✱ Beq rs, rt, L1
 - if (rs == rt) branch to instruction labeled L1;
 - ✱ Bne rs, rt, L1
 - if (rs != rt) branch to instruction labeled L1;
 - ✱ J L1
 - unconditional jump to instruction labeled L1

Compiling If Statements

- C code:
 if (i==j) f = g+h;
 else f = g-h;
 * f, g, ... in \$s0, \$s1, ...
- Compiled MIPS code:
 Bne \$s3, \$s4, Else
 Add \$s0, \$s1, \$s2
 J Exit
 Else: Sub \$s0, \$s1, \$s2
 Exit: ...

Assembler calculates addresses

Compiling Loop Statements

- C code: (with Variable Array Index)

Loop: $g = g + A[i];$

$i = i + j;$

if ($i \neq h$) goto Loop;

Variables g, h, i and j to the registers $\$s1, \$s2, \$s3$ and $\$s4$

Assume A is an array of 100 elements and its base address is in $\$s5$

C code to MIPS code

■ C Code:

$g = g + A[i];$

- * Assume A is an array of 100 elements and its base address is in \$s3
- * Variables g and i to the registers \$s1 and \$s4

C ($g = g + A[i];$) to MIPS

Array A of 100 elements and its base address is in \$s3

Variables G and i to the registers \$s1 and \$s4

- Load $A[i]$ into a temporary register

Due to Byte Addressing, Must multiply i by 4

i.e. $i + i = 2i$ and then $2i + 2i = 4i$

- MIPS Code:

```
add $t1, $s4, $s4    # $t1 = 2 * i
```

```
add $t1, $t1, $t1     # $t1 = 4 * i
```

C ($g = g + A[i];$) to MIPS

Array A of 100 elements and its base address is in \$s3

Variables G and i to the registers \$s1 and \$s4

- To get the address of $A[i]$,
Need to add \$t1 and the base of A in \$s3
i.e., add \$t1, \$t1, \$s3
\$t1=address of $A[i]$ ($4 * i + \$s3$)

C ($g = g + A[i];$) to MIPS

- Now use Load $A[i]$ into a temporary register

i.e., `lw $t0, 0($t1)`

`# $t0 = A[i]`

Final Instruction adds $A[i]$ and g , and places the sum in g :

i.e., `add $s1, $s1, $t0`

`# g = g + A[i]`

C ($g = g + A[i];$) to MIPS

■ MIPS Code:

add \$t1, \$s4, \$s4 # $\$t1 = 2 * i$

add \$t1, \$t1, \$t1 # $\$t1 = 4 * i$

add \$t1, \$t1, \$s3 # $\$t1 = \text{address of } A[i] \text{ (} 4 * i + \$s3 \text{)}$

lw \$t0, 0(\$t1) # $\$t0 = A[i]$

add \$s1, \$s1, \$t0 # $g = g + A[i]$

Compiling Loop Statements

- C code: (with Variable Array Index)

Loop: $g = g + A[i]; \quad i = i + j; \quad \text{if } (i \neq h) \text{ goto Loop};$

Variables g, h, i and j to the registers $\$s1, \$s2, \$s3$ and $\$s4$

Array A of 100 elements and its base address is in $\$s5$

- Compiled MIPS code:

Loop: $\text{add } \$t1, \$s3, \$s3 \quad \# \text{ Temp reg } \$t1 = 2 * i$

$\text{add } \$t1, \$t1, \$t1 \quad \# \text{ Temp reg } \$t1 = 4 * i$

$\text{add } \$t1, \$t1, \$s5 \quad \# \$t1 = \text{address of } A[i]$

$\text{lw } \$t0, 0(\$t1)$

$\text{add } \$s1, \$s1, \$t0 \quad \# g = g + A[i]$

$\text{add } \$s3, \$s3, \$s4 \quad \# i = i + j$

$\text{bne } \$s3, \$s2, \text{Loop}$

Compiling While Loop

- C code:

```
while (save[i] == k)
```

```
    i = i + j;
```

- ★ i in \$s3, j in \$s4, k in \$s5, base address of save in \$s6

- Compiled MIPS code: ?

Probable Solution 1

```
while: add $t1,$s3,$s3;  
      add $t1,$t1,$t1;  
      add $t1,$t1,$s6;  
      lw $t0,0($t1);  
      Beq $t0,$s5,here;  
here: add $s3,$s3,$s4;  
Exit:
```

Probable Solution 2

```
while: add $t1,$s3,$s3;  
      add $t1,$t1,$t1;  
      add $t1,$t1,$s6;  
      lw $t0,0($t1);  
      Beq $t0,$s5,here;  
exit  
here: add $s3,$s3,$s4;  
      J while
```

Probable Solution 3

add t1,s3,s3

add t1,t1,t1

add t0,s4,s4

add t0,t0,t0

add t0,t0,s4

loop: load s1,0(t1)

bnq s1,s5 exit

add t1,t1,t0

j loop

exit:

Probable Solution 4

```
while:  add $t1,$s3,$s3    // Address
        add $t1,$t1,$t1
        add $t1,$t1,$s4    // $s4=Base Addr
        lw $t0, 0($t1)
        Beq $t0, $s5, body // Equality Check
        J Exit
body:   add $s3,$s3,$s4
        J while
Exit : ...
```