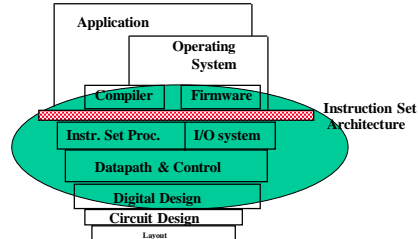
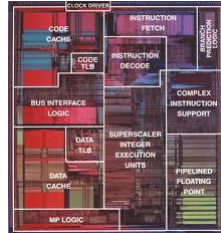




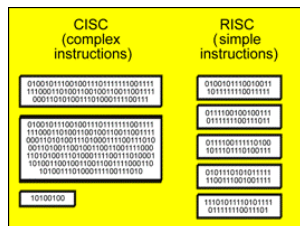
CS/SE 3340

Computer Architecture



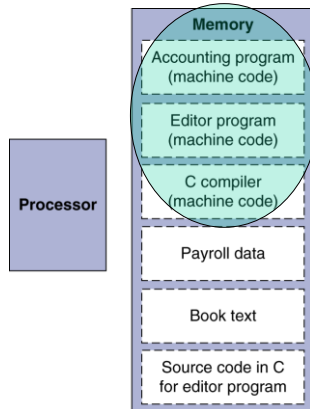
Instructions Representation

Adapted from "Computer Organization and Design, 4th Ed." by D. Patterson and J. Hennessy



MIPS
TECHNOLOGIES

Instructions in Stored Program Computers



- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
 - e.g., compilers, linkers, ...
- Binary compatibility allows compiled programs to work on different computers
 - Standardized ISAs

3

Instruction Set

- The repertoire of instructions of a computer
- Different ISAs have different instruction sets
 - But with many aspects in common
- Early computers had very simple instruction sets
 - Simplified implementation
- Many modern computers also have simple instruction sets

4

CISC and RISC

CISC: Complex Instruction Set Computer

RISC: Reduced Instruction Set Computer

CISC	RISC
Variable length instruction	Single word instruction
Variable format	Fixed-field decoding
Memory operands	Load/store architecture
Complex operations	Simple operations

5

Why RISC?

- CISC architectures were designed under many constraints
 - *Small, slow and expensive memories!* -> compact programs have advantages
 - *On-chip registers are expensive!* -> memory operands
 - *Attempts to bridge the semantic gap* -> high level language features in instructions set
- As technologies advances many of these constraints have become ***less relevant***
- Processor designers wanted to exploit new advances in compiler technology to gain performance
 - Pipelining
 - Cache
- Also to facilitate chip design with *simplicity* and *regularity*

6

The MIPS Instruction Set

- Used as the example throughout the text book
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- Large share of embedded core market
 - Applications in consumer electronics, network/storage equipment, cameras, printers, ...
- Typical of many modern ISAs
- See MIPS Reference Data tear-out card

7

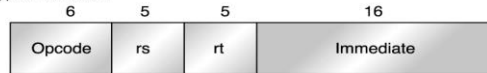
Representing Instructions

- Instructions are encoded in binary
 - Called *machine code*
- MIPS instructions
 - Encoded as *32-bit instruction words*
 - Small number of formats encoding *operation code (opcode)* and *operands (register numbers, ...)*
 - Regularity!
- Register numbers
 - \$t0 – \$t7 are reg's 8 – 15
 - \$t8 – \$t9 are reg's 24 – 25
 - \$s0 – \$s7 are reg's 16 – 23

8

MIPS Instruction Types

I-type instruction



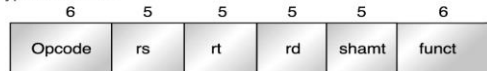
Encodes: Loads and stores of bytes, half words, words, double words. All immediates ($rt \leftarrow rs \text{ op immediate}$)

Conditional branch instructions (rs is register, rd unused)

Jump register, jump and link register

(rd = 0, rs = destination, immediate = 0)

R-type instruction



Register-register ALU operations: $rd \leftarrow rs \text{ funct } rt$

Function encodes the data path operation: Add, Sub, ...

Read/write special registers and moves

J-type instruction

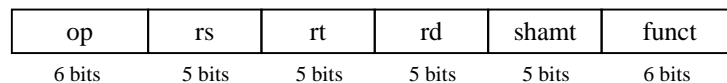


Jump and jump and link

Trap and return from exception

9

MIPS R-format Instructions



- Instruction fields
 - *op*: operation code (opcode)
 - *rs*: first source register number
 - *rt*: second source register number
 - *rd*: destination register number
 - *shamt*: shift amount (00000 for now)
 - *funct*: function code (extends opcode)

10

R-format Example

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

add \$t0, \$s1, \$s2

special	\$s1	\$s2	\$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

00000010001100100100000000100000₂ = 02324020₁₆

11

Arithmetic Operations

- Add and subtract, three operands
 - Two sources and one destination

add a, b, c # a gets b + c
- All arithmetic operations have this form
- *Design Principle 1*: Simplicity favors regularity
 - Regularity makes implementation simpler
 - Simplicity enables higher performance at lower cost

12

Register Operands

- Arithmetic instructions use *register operands*
- MIPS has a 32×32 -bit register file
 - Use for frequently accessed data
 - Numbered 0 to 31
 - 32-bit data called a “word”
- Assembler names
 - \$t0, \$t1, ..., \$t9 for temporary values
 - \$s0, \$s1, ..., \$s7 for saved variables
- *Design Principle 2: Smaller is faster*
 - c.f. main memory: millions of locations

13

Register Operand Example

- C code:
 $f = (g + h) - (i + j);$
 - f, \dots, j in \$s0, ..., \$s4
- Compiled MIPS code:
add \$t0, \$s1, \$s2
add \$t1, \$s3, \$s4
sub \$s0, \$t0, \$t1

14

MIPS I-format Instructions

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- Immediate arithmetic and load/store instructions
 - rt: destination or source register number
 - Constant: -2^{15} to $+2^{15} - 1$
 - Address: offset added to base address in rs
- *Design Principle 4: Good design demands good compromises*
 - Different formats complicate decoding, but allow 32-bit instructions uniformly
 - Keep formats as similar as possible

15

Memory Operands

- Main memory used for composite large data
 - Arrays, structures, dynamic data
- To apply arithmetic operations
 - Load values from memory into registers
 - Store result from register to memory
- Memory is byte addressed
 - Each address identifies an 8-bit byte
- Words are aligned in memory
 - Address must be a multiple of 4
- MIPS supports both Big Endian and Little Endian
 - MARS uses Little Endian

16

Memory Operand Example 1

- 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

- 4 bytes per word

```
lw  $t0, 32($s3)    # load word
add $s1, $s2, $t0
```



17

Memory Operand Example 2

- C code:

`A[12] = h + A[8];`

- `h` in `$s2`, base address of `A` in `$s3`

- Compiled MIPS code:

- Index 8 requires offset of 32

```
lw  $t0, 32($s3)    # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)    # store word
```

18

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
 - More instructions to be executed
- Compiler must use registers for variables as much as possible
 - Only spill to memory for less frequently used variables
 - *Register optimization is important!*

19

Immediate Operands

- Constant data specified in an instruction
`addi $s3, $s3, 4`
- No subtract immediate instruction
 - Just use a negative constant
`addi $s2, $s1, -1`
- *Design Principle 3: Make the common case fast*
 - Small constants are common
 - Immediate operand avoids a load instruction

20

The Constant Zero

- MIPS register 0 (\$zero) is the constant 0
 - Cannot be overwritten
- Useful for common operations
 - E.g., *move between registers*
`add $t2, $s1, $zero`