University of Arizona

CSc 340

Foundations of Computer Systems

Christian Collberg

January 23, 2001

Machine Code

Slide 4–1

Programming in Machine Code

- The topic on computer organization was a bit vague on the details. An instruction was represented as a sequence of four numbers: an operation, two operand registers, and a result register. The last topic showed us how to represent numbers. Now we'll learn how to package them together into instructions.
- Machine code are the numbers that represent computer instructions. We'll look at MIPS machine code in particular.
- The set of valid instructions a given computer is called its instruction set.

Slide 4–2

MIPS Architecture

- Each MIPS instruction is a 32-bit (4-byte) quantity.
- MIPS is a load/store architecture. Most instructions require their operands and results to be in registers. There are two exceptions:
  1. Load instructions: copy data from memory into a register.
  2. Store instructions: copy data from a register into memory.

Slide 4–3

MIPS Architecture . . .

- In general, RISC processors are load/store architectures, and CISC processors are not. CISC processors allow any or all of the operands and results of an instruction to be in memory.
- What implications does this have?
- RISC = Reduced Instructions Set Computer
- CISC = Complex Instructions Set Computer

Slide 4–4

Memory and Registers

- Memory accesses must be properly aligned according to size. Words must be aligned on word boundaries (i.e. the address must be divisable by 4), half-words on half-word boundaries, etc. This includes instructions, which must be aligned on word boundaries.
- MIPS instructions are three-address, meaning an instruction can have up to two operand registers and a destination register. The same register may be used for any or all addresses. Note the confusing use of the term "address", which assumes the CPU is a CISC.

Slide 4–5

Memory and Registers ...

- The MIPS has 32 general-purpose registers, each containing 32 bits.
- General purpose registers can be used to hold data values or addresses, and you can perform any operation on them. There are also special-purpose MIPS registers that are used for special operations.
- The MIPS instruction set is relatively orthogonal, meaning you can mix-and-match general-purpose registers and operations; there are no restrictions.

Slide 4–6

Memory and Registers ...

- Some computers have non-orthogonal instruction sets; certain operations can only be done on certain registers, e.g. multiplication reads its operands from specific registers and writes the result to specific registers. Non-orthogonal instruction sets are typically used because of hardware constraints; they aren't pleasant to program.

Slide 4–7

Registers ...

- MIPS caveats:
  1. Register 0 always contains value 0. You can't change it. It's a good place to get a 0 if you need one, and to store the result of an operation if you don't want it.
  2. Register 31 holds the return address for procedure calls (we'll get to this later).
  3. There are software conventions for using the other registers. This means that the hardware doesn't care what you do with them, but other programmers do. We'll also cover this later.

Slide 4–8

dec	hex	Instruction 31:26	25:21	20:16	15:11	10:6	5:0	Op
0	00	000000	00000	rt	rd	imm5	000000	sll
shift left logical								
0	00	000000	00000	rt	rd	imm5	000010	srl
shift right logical								
0	00	000000	00000	rt	rd	imm5	000011	sra
shift right arithmetic								
0	00	000000	rs	rt	rd	00000	000100	sllv
sll variable								
0	00	000000	rs	rt	rd	00000	000110	srlv
srl variable								
0	00	000000	rs	rt	rd	00000	000111	srav
sra variable								
0	00	000000	rs	00000	00000	00000	001000	jr
jump register								

Slide 4–9

dec	hex	Instruction 31:26	25:21	20:16	15:11	10:6	5:0	Op
0	00	000000	rs	00000	rd	00000	001001	jalr
jump register and link								
0	00	000000	00000	00000	00000	00000	001100	syscall
system call								
0	00	000000	imm20				001101	break
raise exception								
0	00	000000	00000	00000	rd	00000	010000	mfhi
move from hi								
0	00	000000	rs	00000	00000	00000	010001	mthi
move to hi								
0	00	000000	00000	00000	rd	00000	010010	mflo
move from lo								
0	00	000000	rs	00000	00000	00000	010011	mtlo
move to lo								

Slide 4–10

dec	hex	Instruction 31:26	25:21	20:16	15:11	10:6	5:0	Op
0	00	000000	rs	rt	00000	00000	011000	mult
multiply								
0	00	000000	rs	rt	00000	00000	011001	multu
multiply unsigned								
0	00	000000	rs	rt	00000	00000	011010	div
divide								
0	00	000000	rs	rt	00000	00000	011011	divu
divide unsigned								
0	00	000000	rs	rt	rd	00000	100000	add
add								
0	00	000000	rs	rt	rd	00000	100001	addu
add unsigned								
0	00	000000	rs	rt	rd	00000	100010	sub
subtract								

Slide 4–11

dec	hex	Instruction 31:26	25:21	20:16	15:11	10:6	5:0	Op
0	00	000000	rs	rt	rd	00000	100011	subu
substract unsigned								
0	00	000000	rs	rt	rd	00000	100100	and
logical and								
0	00	000000	rs	rt	rd	00000	100101	or
logical or								
0	00	000000	rs	rt	rd	00000	100110	xor
logical xor								
0	00	000000	rs	rt	rd	00000	100111	nor
logical nor								
0	00	000000	rs	rt	rd	00000	101010	slt
set less than								
0	00	000000	rs	rt	rd	00000	101011	sltu
set less than unsigned								

Slide 4–14

dec	hex	Instruction 31:26	25:21	20:16	15:11	10:6	5:0	Op
12	0C	001100	rs	rt	imm16			andi
and immediate								
13	0D	001101	rs	rt	imm16			ori
or immediate								
14	0E	001110	rs	rt	imm16			xori
xor immediate								
15	0F	001111	00000	rt	imm16			lui
load upper immediate								
33	21	100000	rs	rt	imm16			lb
load byte								
34	22	100001	rs	rt	imm16			lh
load half-word								
35	23	100010	rs	rt	imm16			lwl
load word left								

Slide 4–12

dec	hex	Instruction 31:26	25:21	20:16	15:11	10:6	5:0	Op
1	01	000001	rs	00000	imm16			bltz
branch on rs $<$ 0								
1	01	000001	rs	00001	imm16			bgez
branch on rs $\geq$ 0								
1	01	000001	rs	10000	imm16			bltzal
branch on rs $<$ 0 and link								
1	01	000001	rs	10001	imm16			bgezal
branch on rs $\geq$ 0 and link								
2	02	000010	imm26					j
jump								
3	03	000011	imm26					jal
jump and link								
4	04	000100	rs	rt	imm16			beq
branch on rs $=$ rt								

Slide 4–15

dec	hex	Instruction 31:26	25:21	20:16	15:11	10:6	5:0	Op
36	24	100011	rs	rt	imm16			lw
load word								
37	25	100100	rs	rt	imm16			lbu
lb unsigned								
38	26	100101	rs	rt	imm16			lhu
lh unsigned								
39	27	100110	rs	rt	imm16			lwr
load word right								
41	29	101000	rs	rt	imm16			sb
store byte								
42	2A	101001	rs	rt	imm16			sh
store half-word								
43	2B	101010	rs	rt	imm16			swl
store word left								

Slide 4–13

dec	hex	Instruction 31:26	25:21	20:16	15:11	10:6	5:0	Op
5	05	000101	rs	rt	imm16			bne
branch on rs $\neq$ rt								
6	06	000110	rs	00000	imm16			blez
branch on rs $\leq$ 0								
7	07	000111	rs	00000	imm16			bgtz
branch on rs $>$ 0								
8	08	001000	rs	rt	imm16			addi
add immediate								
9	09	001001	rs	rt	imm16			addiu
add immediate unsigned								
10	0A	001010	rs	rt	imm16			slti
set less than immediate								
11	0B	001011	rs	rt	imm16			sltiu
slt immediate unsigned								

Slide 4–18

Sample MIPS machine code

- Here is sample MIPS machine code (in hex) that actually does something:

  `00100100000000010001000000000000000000000`
  `00000010000100001000000000101000000000111`
  `11111111111111`

- To figure out what this program does, we have to decode the instructions. First separate the individual instructions (4 bytes each):

  `00100100000000010001000000000000`
  `00000000000000010000100001000000`
  `00010100000000011111111111111111`

Slide 4–19

Sample MIPS machine code ...

- Group the bits into fields according to the "MIPS Instruction Encodings" document:

```
001001|00000|00001|0001000000000000
op     rs    rt    imm16
```

- The op is the operation to be performed. This is often called the opcode (operation code). In this case the opcode is 0x9, which corresponds to addition immediate without overflow. The "without overflow" means that the values are treated as unsigned binary numbers.
- The result is put in register rt (r1)

Slide 4–16

dec	hex	Instruction 31:26	25:21	20:16	15:11	10:6	5:0	Op
44	2C	101011	rs	rt	imm16			sw
store word								
47	2F	101110	rs	rt	imm16			swr
store word right								

Slide 4–17

MIPS Registers Name	Number	Usage
zero	0	Constant 0
at	$1	Reserved for assembler
v0–v1	$2–$3	Expression evaluation and function results
a0–a3	$4–$7	Argument 1-4
t0–t7	$8–$15	Temporary (not preserved across call)
s0–s7	$16–$23	Saved temporary (preserved across call)
t8–t9	$24–$25	Temporary (not preserved across call)
k0–k1	$26–$27	Reserved for OS kernel
gp	$28	Pointer to global area
sp	$29	Stack pointer
fp	$30	Frame pointer
ra	$31	Return address (used by function call)

Slide 4–20

Sample MIPS machine code . . .

- Register rs (r0) is added to the value in field imm16. imm16 contains a 16-bit immediate value – the value of the number in those 16 bits of the instruction is used as one of the operands. In this instruction imm16 is 0x1000.
- The instruction computes r1 = r0 + 0x1000, which is the same as r1 = 0x1000.

Slide 4–21

Sample MIPS machine code . . .

- Now do the same for the second instruction.

```
000000|00000|00001|00001|00001|000000
op     rs    rt    rd    shamt  func
```

- op = 0. The func field to determine the operation. func = 0 which is shift left logical. This instruction shifts the value in register rt (r1) by the amount in the field shamt (1), and stores the result in register rd (r1).

Slide 4–22

Sample MIPS machine code . . .

- And the last instruction:

```
000101|00000|00001|1111111111111111
op     rs    rt    offset
```

- op = 5, which is branch on not equal. This instruction changes the PC (program counter) by the number of instructions stored in offset (0xFFFF = -1) if the value in rs (r0) does not equal the value in rt (r1). In other words, the PC moves back one instruction if r1 is not zero.

Slide 4–23

Sample MIPS machine code . . .

- In pseudo-code:

```
r1 = 0x1000
do {
   r1 = r1 << 1
} while (r1 != 0)
```

  where `"<<"` means shift left logical, and `"!="` means "does not equal".
- Programming in machine code can be done, but it's (obviously) not pleasant. We'd like to use words, not bits, to describe what's going on. The pseudo-code is much more understandable than the machine code.

Assembly Code

- An assembler is a program that takes a textual representation of a program, and turns them into machine code. For example, we could write the above program as:

```
addiu r1, r0, 0x1000
sll   r1, r1, 1
bne   r0, r1, -1
```

- This is much easier to understand than machine code (although still a bit cryptic), and can be converted into machine code by a simple assembler.

Readings and References

- Read Maccabe, section 3.1.4, pp. 80–85.