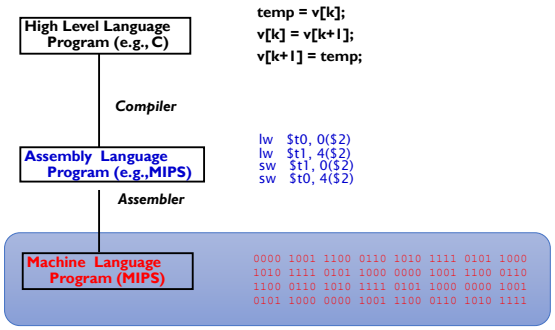
Lecture 07: Instructions as Numbers

(CPEG323: Intro. to Computer System Engineering)

Levels of Program Code



Assembly vs. machine language

- MIPS is an assembly language.
  - We assign names to operations (e.g., add) and operands (e.g., $t0).
  - Branches and jumps use labels instead of actual addresses.
  - Assemblers support many pseudo-instructions.
- Programs must eventually be translated into machine language, a binary format that can be stored in memory and executed by the CPU.

MIPS instructions

- MIPS is designed to be easy to fetch and decode:
  - Each instruction is the same length, 32 bits
  - Only three different instruction formats, with many similarities
  - Format determined by its first 6 bits: operation code, or *opcode*
- Fixed-size instructions:
  (+) easy to fetch/pre-fetch instructions
  (–) limits number of operations, limits flexibility of ISA
- Small number of formats:
  (+) easy to decode instructions (simple, fast hardware)
  (–) limits flexibility of ISA

Question

- Which are represented by 0x00494824 ?
  - the integer 4802596
  - the string "$HI"
  - the float 6.7298704e-39
  - the instruction and $9, $2, $9

Answer

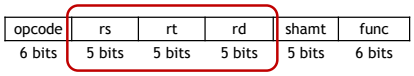
- Which are represented by 0x00494824 ?
  Answer: All of them. They are just different interpretations of the same bit patterns.
- How does the machine know which interpretation you want?
  You have to explicitly tell the machine which interpretation you want.
  - Use an integer load (lw) to interpret it as an int
  - Use a floating point load (l.s) to interpret it as a float
  - Use a branch or a jump (bne or j) to interpret it as an instruction

Instructions as Numbers

- Instructions are also kept as binary numbers in memory
  - Stored program concept
  - As easy to change programs as it is to change data
- Register names mapped to numbers
- Need to map instruction operation to a part of number

Instruction Format (1) R-Format

- Register-to-register arithmetic instructions use the R-type format



- Six different fields:
  - opcode is an operation code that selects a specific operation
  - rs and rt are the first and second source registers
  - rd is the destination register
  - shamt is only used for shift instructions (sll, srl, sra)
  - func is used together with opcode to select an arithmetic instruction

MIPS registers eocoding

- We have to encode register names as 5-bit numbers from 00000 to 11111
  - e.g., $t8 is register $24, which is represented as 11000
- The number of registers available affects the instruction length:
  - R-type instructions references 3 registers: total of 15 bits
  - Adding more registers either makes instructions longer than 32 bits, or shortens fields like opcode (reducing number of available operations)

An Example of R-Format

- MIPS Instruction: **add $8,$9,$10**

opcode = 0
funct = 32
rd = 8 (destination)
rs = 9 (first operand)
rt = 10 (second operand)
shamt = 0 (not a shift)

Decimal number per field representation:

0	9	10	8	0	32

Binary number per field representation:

000000	01001	01010	01000	00000	100000

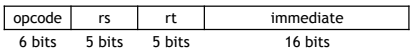
hex

How to encode instructions with immediates?

- Note that a 5-bit field only represents numbers up to the value 31, but immediates may be much larger than this
- We need a new instruction format that is partially consistent with R-format:
  - If instruction has immediate, then it uses at most 2 registers.

Instruction Format (2) I-Format

- Used for immediate instructions, plus load, store and branch



- For uniformity, opcode, rs and rt are located as in the R-format
- The meaning of the register fields depends on the exact instruction:
  - rs is *always* a source register (memory address for load and store)
  - rt is a *source* register for store and branch, but a *destination* register for all other I-type instructions
- The immediate is a 16-bit signed two's-complement value.
  - It can range from –32,768 to +32,767.
  - Question: How does MIPS load a 32-bit constant into a register?
  - Answer: Two instructions. Make the common case fast.

What if an immediate is more than 16 bits?

- Larger constants can be loaded into a register 16 bits at a time.
  - The load upper immediate instruction lui loads the highest 16 bits of a register with a constant, and clears the lowest 16 bits to 0s.
  - An immediate logical OR, ori, then sets the lower 16 bits.
- To load the 32-bit value 0000 0000 0011 1101 0000 1001 0000 0000:

```
lui $s0, 0x003D        # $s0 = 003D 0000 (in hex)
ori $s0, $s0, 0x0900   # $s0 = 003D 0900
```

Branche Instructions

- For branch instructions, the constant field is not an address, but an *offset* from the current program counter (PC) to the target address.

```
      beq  $t0, $t1, EQ
      add  $t0, $t0, $t1
      addi $t1, $t0, $0
EQ:   add  $v1, $v0, $v0
```

- Since the branch target EQ is *two* instructions past the instruction after the beq, the address field contains 2

000100	10001	10010	0000 0000 0000 0010
op	rs	rt	address (offset)

14

Addresses in Branch Instructions: PC-relative

- Given an instruction "beq $t1,$t2, immediate" and the PC, what is the address for the next instruction?
- Answer:
  - If we don't take the branch:
    PC = PC + 4
  - If we do take the branch:
    PC = (PC + 4) + (immediate * 4)
  - Observations
    - Immediate field specifies the number of words to jump, which is simply the number of instructions to jump.
    - Immediate field can be positive or negative

Instruction Format (3) J-Format

- The jump instructions (e.g., j and jal) use J-type instruction format.

opcode	address (exact)
6 bits	26 bits

- The jump instruction contains a word address, not an offset.
  - Remember that each MIPS instruction is one word long, and word addresses must be divisible by four.
  - Instead of saying "jump to address 4000,", it is enough to just say "jump to instruction 1000."
  - only the top 26 bits actually stored (last two are always 0)
  - Take the 4 highest order bits from the PC
- For even longer jumps, the jump register (jr) instruction can be used.

```
jr $ra    # Jump to 32-bit address in register $ra
```

16

Addresses in Jump Instructions

- What is the address of the target instruction for "j label"?
- Answer:
  - New PC = { (PC+4)[31..28], target address, 00 }
  - Note: { , , } means concatenation
  - { 4 bits , 26 bits , 2 bits } = 32 bit address
  - For example, given the first four bits of PC is 1010, and the label = 11111111111111111111111111,
    - So, the new PC value is 10101111111111111111111111111100

Question

Which instruction has same representation as 35_{ten}?

1. add $0, $0, $0
2. subu $s0,$s0,$s0
3. lw $0, 0($0)
4. addi $0, $0, 35
5. subu $0, $0, $0

opcode	rs	rt	rd	shamt	funct
opcode	rs	rt	rd	shamt	funct
opcode	rs	rt	offset		
opcode	rs	rt	immediate		
opcode	rs	rt	rd	shamt	funct

Registers numbers and names:
0: $0, .. 8: $t0, 9:$t1, ..15: $t7, 16: $s0, 17: $s1, .. 23: $s7

Opcodes and function fields:
add: opcode = 0, funct = 32
subu: opcode = 0, funct = 35
addi: opcode = 8
lw: opcode = 35

Answer

Which instruction bit pattern = number 35?

1. add $0, $0, $0	0	0	0	0	0	32
2. subu $s0,$s0,$s0	0	16	16	16	0	35
3. lw $0, 0($0)	35	0	0			0
4. addi $0, $0, 35	8	0	0			35
5. subu $0, $0, $0	0	0	0	0	0	35

Registers numbers and names:
0: $0, …, 8: $t0, 9:$t1, …,16: $s0, 17: $s1, …,

Opcodes and function fields
add: opcode = 0, function field = 32
subu: opcode = 0, function field = 35
addi: opcode = 8
lw: opcode = 35

An Example

A[300] = h + A[300];

lw $t1,1200($t0)
addu $t1,$s2,$t0
sw $t1,1200($t0)

$t0 - $8;
$t1 - $9;
$s2 - $18
addu – funt: 33; opcode: 0.
lw – opcode: 35
sw – opcode: 43

R-type	op	rs	rt	rd	shamt	funct
I-type	op	rs	rt	address or constant		
J-type	op	address				

An Example (Cont.)

A[300] = h + A[300];

lw $t0,1200($t1)
addu $t1,$s2,$t0
sw $t0,1200($t1)

35	9	8	1200			
0	18	8	9	0	33	
43	9	8	1200			

$t0 - $8;
$t1 - $9;
$s2 - $18
addu – funt: 33; opcode: 0.
lw – opcode: 35
sw – opcode: 43

R-type	op	rs	rt	rd	shamt	funct
I-type	op	rs	rt	address or constant		
J-type	op	address				

A More Complicated Example

Address	Loop:							
800	sll $t1,$s3,2							
804	addu $t1,$t1,$s6							
808	lw $t0,0($t1)							
812	bne $t0,$s5, Exit							
816	addi $s3,$s3,1							
820	j Loop							
	Exit:							

$t0-$t7: $8-$15
$s0-$s7: $16-$23
addu – funt: 33; opcode: 0.
lw – opcode: 35
sll – funt:0; opcode:0;
j – opcode=2;
bne – opcode=4;
addi – opcode=8;

R-type	op	rs	rt	rd	shamt	funct
I-type	op	rs	rt	address or constant		
J-type	op	address				

A More Complicated Example (Cont.)

Address	Loop:							
800	sll $t1,$s3,2	R	0	0	19	9	2	0
804	addu $t1,$t1,$s6	R	0	9	22	9	0	33
808	lw $t0,0($t1)	I	35	9	8	0		
812	bne $t0,$s5, Exit	I	5	8	21	2		
816	addi $s3,$s3,1	I	8	19	19	1		
820	j Loop	J	2	200				
	Exit:							

R-type	op	rs	rt	rd	shamt	funct
I-type	op	rs	rt	address or constant		
J-type	op	address				

Summary

- In computers, instructions are stored as data.
- MIPS Instruction Format
  - I-format: used for instructions with immediates, lw and sw (since the offset counts as an immediate), and the branches (beq and bne)
  - J-format: used for j and jal
  - R-format: used for all other instructions

Decoding Machine Language

- How to convert 0/1 strings back to C code?
- For each 32 bits:
  1. Look at opcode to determine the instruction format.
  2. Split 32 bits into different fields based on the corresponding instruction format.
  3. Mapping the values in each field to register names, labels, etc.
  4. Convert the MIPS code to C code.

An example of instruction decoding

- Given six machine language instructions in hexadecimal:

  00001025_{hex}
  $0005402A_{hex}$
  11000003_{hex}
  00441020_{hex}
  $20A5FFFF_{hex}$
  08100001_{hex}

- Assume that the first instruction is at address $4{,}194{,}304_{ten}$ ($0x00400000_{hex}$).

Step 1: Convert to binary

- The six machine language instructions in binary:

```
00000000000000000001000000100101
00000000000001010100000000101010
00010001000000000000000000000011
00000000010001000001000000100000
00100000101001011111111111111111
00001000000100000000000000000001
```

Step 2: Identify the Format based on Opcode

Format:

```
R 000000 00000 00000 00010 00000 100101
R 000000 00000 00101 01000 00000 101010
I 000100 01000 00000 0000000000000011
R 000000 00010 00100 00010 00000 100000
I 001000 00101 00101 1111111111111111
J 000010 00000100000000000000000001
```

Opcode:
0 means R-Format,
2 or 3 mean J-Format,
otherwise I-Format.

Step 3: Separating Fields

Format:

R	0	0	0	2	0	37
R	0	0	5	8	0	42
I	4	8	0	+3		
R	0	2	4	2	0	32
I	8	5	5	-1		
J	2	1,048,577				

Step 4: Writing the Assemebly Code

Address: Assembly instructions:

```
0x00400000      or    $2,$0,$0
0x00400004      slt   $8,$0,$5
0x00400008      beq   $8,$0,3
0x0040000c      add   $2,$2,$4
0x00400010      addi  $5,$5,-1
0x00400014      j     0x100001
```

```
        or    $v0,$0,$0
Loop:   slt   $t0,$0,$a1
        beq $t0,$0,Exit
        add   $v0,$v0,$a0
        addi  $a1,$a1,-1
        j     Loop
Exit:
```

Step 4: MIPS → C

```
        or    $v0,$0,$0
Loop:   slt   $t0,$0,$a1
        beq $t0,$0,Exit
        add   $v0,$v0,$a0
        addi  $a1,$a1,-1
        j     Loop
Exit:
```

```
product = 0;
while (multiplier > 0) {
    product += multiplicand;
    multiplier -= 1;
}
```

$v0: product
$a0: multiplicand
$a1: multiplier

Reading

- 5th Edition: 2.5

33