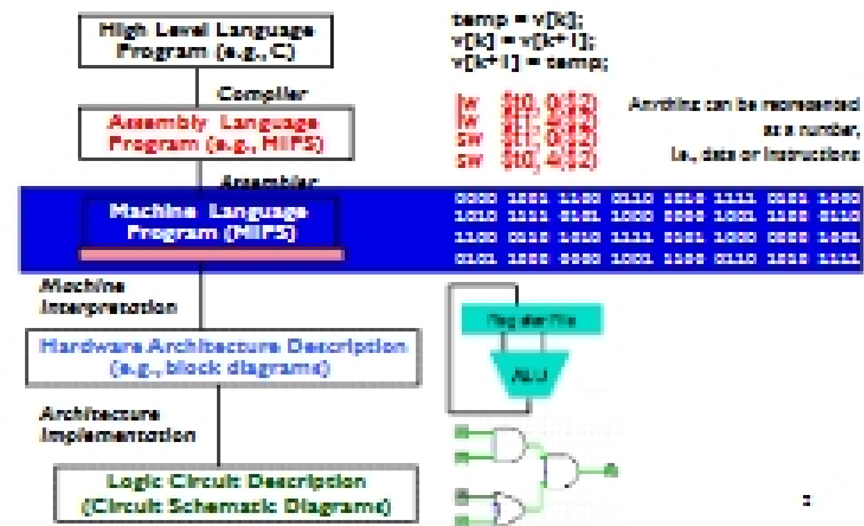
Lecture 07: Instructions as Numbers

(CPEG323: Intro. to Computer System Engineering)

1

Levels of Representation



2

Assembly vs. machine language

- So far we've been using 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 decoded by the CPU.

MIPS instructions

- MIPS machine language is designed to be easy to fetch and decode:
  - each MIPS 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
- Studying MIPS machine language will also reveal some restrictions in the instruction set architecture, and how they can be overcome.

4

Question

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

5

Answer

- Which of the following are represented by the hexadecimal number 0x00494824 ?

  Answer: All of them. They are just different interpretations of the same bit patterns.
  (note: the string representation depends on endianness)

- Then 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 them as an int
  - Use a floating point load (l.s) to interpret them as a float
  - Use a branch or a jump (bne or j) to interpret them as an instruction

6

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

R-type format

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

opcode	rs	rt	rd	shamt	func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- 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

- 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)

opcode	rs	rt	rd	shamt	func

R-Format Example (1/2)

- MIPS Instruction:

```
add   $8,$9,$10
```

opcode = 0 (look up in table in book)

funct = 32 (look up in table in book)

rd = 8 (destination)

rs = 9 (first *operand*)

rt = 10 (second *operand*)

shamt = 0 (not a shift)

R-Format Example (2/2)

- **MIPS Instruction:**

```
add   $8,$9,$10
```

Decimal number per field representation:

0	9	10	8	0	32

Binary number per field representation:

000000	01001	01010	01000	00000	100000

hex

What about immediates?

- What about instructions with immediates?
  - 5-bit field only represents numbers up to the value 31: immediates may be much larger than this
  - Ideally, MIPS would have only one instruction format (for simplicity): unfortunately, we need to compromise
- Define new instruction format that is partially consistent with R-format:
  - First notice that, if instruction has immediate, then it uses at most 2 registers.

I-type 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.

13

Larger constants

- 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:

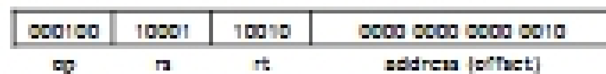
```
lui $s0, 0x003D          # $s0 = 003D 0000 (in hex)
ori $s0, $s0, 0x0900     # $s0 = 003D 0900
```

- This illustrates the principle of making the common case fast.
  - Most of the time, 16-bit constants are enough.
  - It's still possible to load 32-bit constants, but at the cost of two instructions and one temporary register.

Branches

- Two branch instructions:

```
beq  $t0, $t1, label   # if t0 == t1, jump to "label"
bne  $t0, $t1, label   # if t0 != t1, jump to "label"
```



- 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

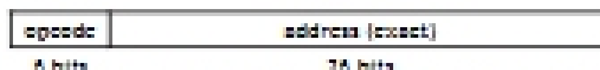
15

Branches: PC-relative addressing

- Branch Calculation:
  - If we don't take the branch:
    PC = PC + 4 = byte address of next instruction
  - 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

J-type format

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



- 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

17

J-type format

- Summary:
  - New PC = { (PC+4)[31..28], target address, 00 }
- Understand where each part came from!
- Note: { , , } means concatenation
  { 4 bits , 26 bits , 2 bits } = 32 bit address
  - { 1010, 11111111111111111111111111, 00 } = 10101111111111111111111111111100
  - Note: Book uses ||