Lecture 3: Introduction to MIPS

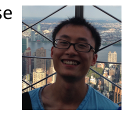
(CPEG323: Intro. to Computer System Engineering)

1

Annoucement

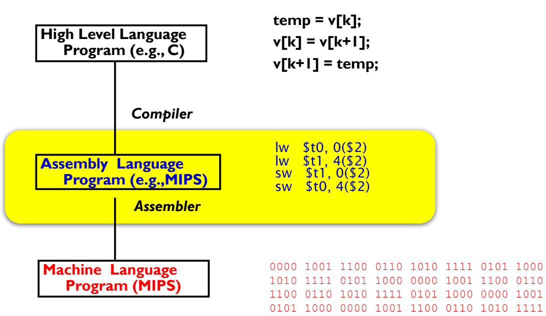
- MP1 out
- Lecture plans for the next two weeks
  - September 10 to 14: number representation
    - By TA Fateme

  - September 17 to 19: c programming (bit wise operation)
    - By TA Kuang



  - September 21: MIPS (cont.)

Levels of Program Code



Assembly Language

- A language that computers can understand
- **Instructions**
  - – CPU's primitives operations
  - E.g., words that a computer can understand
- **Instruction Set Architecture**
  - Vocabulary that a computer can understand
  - E.g., MIPS, ARM, Intel x86, RISC-V, …,
- CPU belong to families based on the ISAs.
  - E.g., iPhone's is different from the Mac's, but same as the iPad's.

Instruction set architectures

- The ISA is an interface between software and hardware
  —the hardware "promises" to implement all ISA instructions
  —the software uses ISA primitives to build complex programs
- The instruction set architecture affects the hardware design
  —simple ISAs require simpler, cheaper processors
- Also affects software design
  —simple ISAs result in longer programs

5

History of ISA

- Early trend was to add more and more instructions to new CPUs to do elaborate operations.
  - e.g., VAX architecture had an instruction to multiply polynomials
- RISC (Reduced Instruction Set Computing) design principle:
  - Keep the instruction set small and simple, make it easier to build fast hardware
  - Let software do complicated operations by composing simpler ones.

6

RISC Design Principles

- A number of the more common strategies include:
  - Fixed instruction length, generally a single word;
  - Simplified addressing modes;
  - Fewer and simpler instructions in the instruction set;
  - Only load and store instructions access memory;
  - Let the compiler do it.

7

MIPS - Instruction Set in CPEG323

- MIPS: semiconductor company that built one of the first commercial RISC architecture (1984-2013).
- Advantages:
  - An elegant example of *RISC* architectures.
  - Real, yet simple.
  - Still used in many places, primarily in embedded systems such as routers and game devices.

8

What you will need to learn for Exams

- You must become "fluent" in MIPS assembly:
  - Translate from C to MIPS and MIPS to C
- Example: Translate the following recursive C function into MIPS

```
int pow(int n, int m) {
  if (m == 1)
    return n;
  return n * pow(n, m-1);
}
```

How are arguments passed?

How are values returned?

How are complex expressions broken into simple instructions?

9

Registers – Assembly Variables

- High-Level Programming languages
  - variables (the number could vary, and be very large)
- Assembly languages
  - registers (fixed, smaller number)
- MIPS Instruction Set has 32 registers, each of which holds a 32-bit value.
  - Benefit:
    - Fast (because they are directly in hardware)
  - Drawback:
    - Need to be careful with the efficient use of registers (because of the limited number)

10

MIPS register names

- MIPS register names begin with a $. There are two naming conventions:
  - by number: $0 $1 $2 … $31
  - by (mostly) two-character names, such as:

    $a0-$a3 $s0-$s7 $t0-$t9 $sp $ra

- Not all of the registers are equivalent:
  - e.g., register $0 or $zero always contains the value 0
  - some have special uses, by convention ($sp holds "stack pointer")
- You have to be a little careful in picking registers for your programs
  - for now, stick to the registers $t0-$t9

11

MIPS Arithmetic Operations

12

MIPS: register-to-register, three address

- MIPS is a register-to-register, or load/store, architecture
  - Destination and sources of instructions must all be registers
- MIPS uses three-address instructions for data manipulation
  - Each ALU instruction contains a destination and two sources.
  - For example, an addition instruction (a = b + c) has the form:

```
operation      operands
add       a,      b,      c
      destination   sources
```

13

Basic arithmetic and logic operations

- The basic integer arithmetic operations include the following:

```
add     sub     mul     div
```

- And here are a few bitwise operations:

```
and     or     xor     nor
```

- Remember that these all require three register operands; for example:

```
add $t0, $t1, $t2   # $t0 = $t1 + $t2
mul $s1, $s1, $a0   # $s1 = $s1 x $a0
```

Note: a full MIPS ISA reference can be found in Appendix A (linked from website)

14

Larger expressions

- Complex arithmetic expressions may require multiple MIPS operations
- Example: t0 = (t1 + t2) × (t3 – t4)

```
add   $t0, $t1, $t2     # $t0 contains $t1 + $t2
sub   $t6, $t3, $t4     # temp value $t6 = $t3 - $t4
mul   $t0, $t0, $t6     # $t0 contains the final product
```

- Temporary registers may be necessary, since each MIPS instructions can access only two source registers and one destination
  - in this example, we could re-use $t3 instead of introducing $t6
  - must be careful not to modify registers that are needed again later

15

How are registers initialized?

- Special MIPS instructions allow you to specify a signed constant, or "immediate" value, for the second source instead of a register
  - e.g., here is the immediate add instruction, addi:

```
addi $t0, $t1, 4        # $t0 = $t1 + 4
```

- Immediate operands can be used in conjunction with the $zero register to write constants into registers:

```
addi   $t0, $0, 4       # $t0 = 4
```

Shorthand: li $t0, 4 (pseudo-instruction) # $t0 = 4

16

Our first MIPS program

- Let's translate the following C++ program into MIPS:

```
void main() {
  int i = 516;
  int j = i*(i+1)/2;
  i = i + j;
}
```

```
main:                       # start of main
    li    $t0, 516          # i = 516
    addi  $t1, $t0, 1       # i + 1
    mul   $t1, $t0, $t1     # i * (i + 1)
    li    $t2, 2
    div   $t1, $t1, $t2     # j = i*(i+1)/2
    add   $t0, $t0, $t1     # i = i + j

    jr    $ra               # return
```

17

Data Transfer

18

What if we need more space?

- Data can be stored in the memory and then transferred to/from the registers.

- **Data transfer instructions:**
  - Memory to register: lw, lb
  - Register to memory: sw, sb

How to Specify Source and Destination?

- **Register:**
  - specified by the numbers or the names
- **Memory address:**
  - Memory can be seen as a single one-dimensional array.
  - MIPS memory is byte-addressable: each memory address references an 8-bit quantity.

Address	0	1	2	3	4	5	6	7	8	9	10	
8-bit data												. . .

  - A memory address can be specified as the sum of the following two values:
    - A register containing a pointer to memory
    - A numerical offset (in bytes)
    - For example, 8($t0) specifies the memory address pointed to by the value in $t0 plus 8 bytes.

Transfer from Memory to Register

Data flow

- MIPS instructions: **lw** and **lb** (load word or load byte)

- Example: lw $t0,12($s0)
  - It computes the memory address by summing up the value in $s0 and 12, reads the value from the computed memory address, and stores the value into register $t0

Transfer from Register to *Memory*

Data flow

- MIPS instruction: **sw** and **sb** (store word and store byte)

- Example: sw $t0,12($s0)

  It read the value from register $t0 and then store in memory. The memory address is computed by adding the values in $t0 and 12.

Example MIPS code using data transfer

```
.data
from: .byte 1
to: .byte 0

.text
main:
 la    $t0, from
 lb  $t1, 0($t0)
 la  $t0, to
 sb  $t1, 0($t0)
```

```
char from = 1, to = 0;

void main() {
    to = from;
}
```

Anther Example

- Assume A is an array of 100 words, variables g and h map to registers $s1 and $s2, the starting address, or base address, of the array A is in $s3

  A[10] = h + A[3];
- Turns into

  lw $t0,12($s3) # Temp reg $t0 gets A[3]

  add $t0,$s2,$t0 # t0 = h + A[3]

  sw $t0,40($s3) # A[10] = h + A[3]