Assembly (Slides T2-T3)

VE370FA22 TA Yuxuan Tang

Levels of Program Code

- High-level language (translator: compiler)
- Assembly language (translator: assembler)
- Machine language

Instruction Set (ISA)

- A collection of instructions that a computer understands
- Different computers have different instruction sets
- Types:
  - Reduced Instruction Set Computer – RISC
  - Complex Instruction Set Computer – CISC
- We will base our discussions on **RISC-V 32-bit ISA** in this course

Design Principle

- Simplicity favors regularity
- Smaller is faster
- Make the common case fast

Operands

Register operands

RISC-V RV32 has a 32 × 32-bit register file, from x0 to x31

- `x0` : the constant value 0
- `x1 (ra)` : return address
- `x2 (sp)` : stack pointer
- `x3 (gp)` : global pointer
- `x4 (tp)` : thread pointer
- `x5 - x7` , `x28 - x31` : temporaries
- `x8` : frame pointer
- `x9` , `x18 - x27` : saved registers
- `x10 - x11` : function arguments/results
- `x12 - x17` : function arguments

e.g.
c code:

```
a = b + c;
```

Assume a,b and c are put in `x5` , `x6` and `x7` respectively.Then the corresponding assembly is

```
add x7, x5, x6
```

Memory operands

Memory organization

- RISC-V memory is **byte addressable**. Each 8-bit byte has a unique address.
- A word has 4 bytes. A word address must be integer multiples of 4, i.e. the last digit of a word address must be 0, 4, 8, C.
- ISC-V memory is Little Endian

Big/Little Endian

Little Endian: Least-significant byte at smallest byte address of a word.

e.g. A word 0x12345678

Endian	0x0fff0000	0x0fff0001	0x0fff0002	0x0fff0003
Little	78	56	34	12
Big	12	34	56	78

Array in Memory

Address of Array = Base Address + Offset = Base Address + (index × 4)
`&A[n] = &A[0] + 4n` in RV32(4 bytes/word)
`&A[n] = &A[0] + 8n` in RV64(8 bytes/word)

Steps to use memory operands

- Load values from memory into registers
- Perform arithmetic operations with registers
- Store result from register to memory

e.g.
c code:

```
A[i] = B[j] + 5;
```

Assume i, j are in `x5`, `x6` respectively, and base address of array `A` and `B` are in `x7`, `x28` respectively.Then the corresponding assembly is

```
slli x5, x5, 2
add x5, x5, x7
slli x6, x6, 2
add x6, x6, x28
lw x29, 0(x6)
addi x29, x29, 5
sw x29, 0(x5)
```

Immediate operands (constant)

- Involves constant data, like 0, 1, 2… or -1, -2…
- A useful constant: x0 = 0. Can not be overwritten and can be used to clear a register.
- `addi`, `slli`… No **`subi`** !

Get Familiar with Other Operations!

Logical

- `and` , `andi` , `or` , `ori` , `xor` , `xori`
- `sll` , `slli` , `srl` , `srli` shift left/right logical – Fill vacated bits with 0 bits.
- `sra` , `srai` shift right arithmetic – Fill vacated bits with sign bit

Arithmetic

`add` , `addi` , `sub`

Conditional

`beq` , `bne` , `blt` , `bge`
Branch if the condition holds.
e.g.

```
beq x5, x6, ELSE
add x5, x0, x0
ELSE: ...
```

Meaning: If `x5 == x6` , neglect the code in the second line and execute the code after `ELSE:`.

Load/Store

`sw` , `lw` , `lb` , `lbu` , `lh` , `sb` ...
e.g.

```
lb x5, 2(x6)
```

Meaning: Load the byte stored in `Memory[x6+2]` into `x5` and sign extend to 32 bits.
e.g.

```
sb x5, 40(x6)
```

Meaning: Store just lowest byte in `Memory[x6+40]`. No extension!!!

Jump

`jal`

```
jal x1, Label1
```

Meaning: `x1` increases by 4 so that it becomes return address register. Program counter points to where the `Label1` represents.

`jalr`

```
jalr x5, offset(x1)
```

Meaning: Program counter points to where `offset+x1` represents (Usually the offset is 0), and x0 become pc+4.
In Ripes, there are some differences in syntax of `jalr`. The two kinds of syntax supported by Ripes are

```
jalr x5, x1(offset)
jalr x5, x1, offset
```

You can just use the following instruction for jump.

```
jr x1
```

Meaning: Program counter points to where `x1` represents.

Load Upper Immedaite

`lui`

```
lui x5, 0x12345
```

Meaning: copy `0x12345` (20-bit constant) to bits `[31:12]` of register `x5`.

Program Counter (PC)

- A special register that points to the instruction to be executed next
- Each instruction is encoded as a 32-bit word

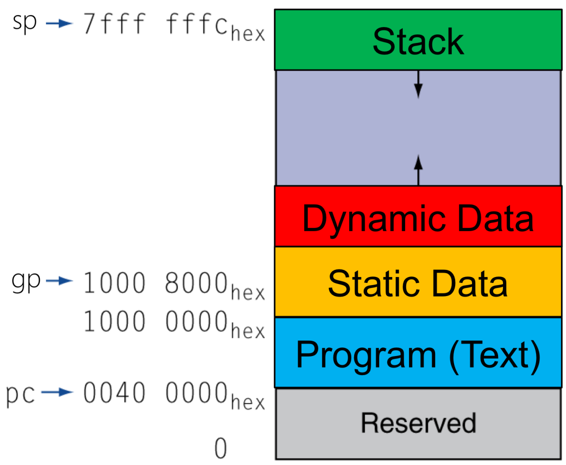
- PC increase by 4 when go to fetch the next instruction

Memory Layout

- **Stack Pointer**: Used when you need to reserve some space in stack to store the important variables. Growing towards low address.
  e.g.

```
addi sp, sp, -12
...
# Several operations
...
addi sp, sp, 12
```

- **global pointer**: initialized to the middle of Static data, 0x10008000 allowing ±offset.
- Text: program code
- Static data: global/static variables
- Dynamic data: heap
- Stack: storage for temporary variable in functions



Function Calling

General Steps

1. Place parameters in registers x10 to x17
2. Call function and transfer control to function
3. Acquire storage on stack for the function
4. Save (push) important registers on the stack
5. Perform function’s operations

6. Place result in register x10 and x11 for caller
7. Restore (pop) important registers from the stack
8. Return storage on stack
9. Return to the place of function call (using x1)

Leaf Functions

- Functions that do not call other functions
- Only save **saved registers** (`x8` , `x9` , `x18-x27`)

Non-Leaf Functions

- Functions call other functions
- Before calling other functions, make sure you save its return address (`x1`), argument registers (`x10` , `x11` …), and temporary registers needed after calling functions returned (`x5` , `x6` …)

Function Examples

*Please try all the examples in the lecture slides by yourselves!!!

Loop

c code:

```c
int add(int *a, int size) {
    //REQUIRES: size is positive integers
    int result = 0;
    for (int i = 0; i < size; i++) {
        result = result + a[i];
    }
    return result;
}
```

Assume two arguments `a` and `size` are stored in `x11` and `x12` respectively. And the returnedresult is stored in `x10`.
assembly:

```
ADD:
    addi x10, x0, 0 # Initialize the result
    add x5, x0, x11 # Initialize the address of the first word
    addi x6, x0, 0 # Initialize the counter (i)
LOOP:
    lw x7, 0(x5) # Load a[i]
    add x10, x10, x7
    addi x6, x6, 1
    addi x7, x7, 4
    bne x6, x12, LOOP
    jr x1
```

Recursion

c code:

```c
int fact (int n) {
    //REQUIRES: n is a positive integer
    if (n < 3) return n;
    else return n * fact(n-1);
}
```

assembly:

```
fact:
    addi sp, sp, -8
    sw x1, 4(sp)
    sw x10, 0(sp)
    addi x5, x10, -3
    bge x5, x0, L1
    addi sp, sp, 8 # Why no need to restore x1 and x10?
    jr x1
L1:
    addi x10, x10, -1
    jal x1, fact
    addi x6, x10, 0 # Now what is the value stored in x10
    lw x10, 0(sp)
    lw x1, 4(sp)
    addi sp, sp, 8
    mul x10, x10, x6
    jr x1
```

Reference

[1] VE370 FA22 Slides T2

[2] VE370 SU22 Slides T3

[3] VE370SU22_RC1