COMPSCI 2GA3 Tutorial 2 Note

Note:

This note does NOT cover all the materials in Chapter 1 -- Only the formulas rated to sample questions of this tutorial are included. Therefore, you may want to make yourself more familiar with things that are presented in lectures.

For any questions about the tutorials and courses, feel free to contact me. (Email: wangm235@mcmaster.ca)

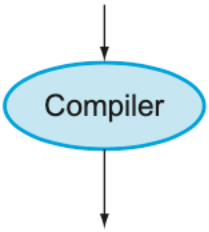
GLHF :)

Mingzhe Wang

The Big Picture

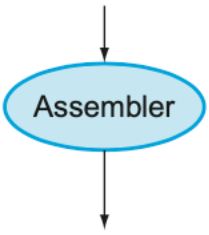
High-level language program (in C)

```
swap(size_t v[], size_t k)
{
   size_t temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```



Assembly language program (for RISC-V)

```
swap:
      slli x6, x11, 3
      add  x6, x10, x6
      ld   x5, 0(x6)
      ld   x7, 8(x6)
      sd   x7, 0(x6)
      sd   x5, 8(x6)
      jalr x0, 0(x1)
```



32 bits

Binary machine language program (for RISC-V)

```
00000000001101011001001100010011
00000000011001010000001100110011
00000000000000011001001010000011
00000000100000011001001110000011
00000000011100110011000000100011
00000000010100110011010000100011
00000000000000000100000001100111
```

The **instruction set** of a computer is its repertoire of instructions that it can perform. ISA defines the interface between hardware and software.

Many modern computers now also have simple instruction sets called Reduced Instruction Set Computers (RISC).

The RISC-V Instruction Set

- Developed at UC Berkeley starting in 2010 as open ISA, RISC-V instructions are 32 bits [31:0].

Some facts about RISC-V

2. Instruction Set Architecture

"Ideally, your initial instruction set should be an exemplar, …"

- Instruction set architecture (ISA) defines the interface between the hardware and software
- instruction set is the language of the computer
- RISCV instructions are 32-bits, instruction[31:0]

- RISC-V assembly[1] language notation
  - uses 64-bit registers, 64-bits refer to double word, 32-bits refers to word (8-bits is byte).
  - there are 32 registers, namely *x0-x31*, where *x0* is always zero
  - to perform arithmetic operations (add, sub, shift, logical) data must always be in registers
    - the number of variables in programs is typically larger than 32, hence 'less frequently used' [or those used later] variables are 'spilled' into memory [spilling registers]
    - registers are faster and more energy efficient than memory
    - for embedded applications where code size is important, a 16-bit instruction set exists, RISC-V compressed (e.g. others exist also ARM Thumb and Thumb2)

  - byte addressing is used, little endian (where address of 64-bit word refers to address of 'little' or rightmost byte, [containing bit 0 of word]) so sequential double word accesses differ by 8 e.g. byte address 0 holds the first double word and byte address 8 holds next double word. (Byte addressing allows the supports of two byte instructions)
  - memory contains 2^{61} memory words - using load/store instructions e.g. 64-bits available (bits 63 downto 0, 3 of those bits are used for byte addressing, leaving 61 bits)

(source: https://ece.uwaterloo.ca/~cgebotys/NEW/ECE222/index.htm)

Some points that can easily raise confusion

1 byte = 8 bits ,1 word = 32 bits, 1 doubleword = 64 bits

Instructions are 32-bits, instruction[31:0]

Byte-addressing!!!!!!!!!!!!!!!!!!!!!!!!

64-bit register/memory/data for our textbook (all codes in our textbook)

32-bit register/memory/data for venus (all code in an online RISC-V simulator)

(This could affect our assembly code. So you would like to know which is assumed! But no worry, they follow the same logic :))

Memory

(We omit the “Endians” part, because this could raise unnecessary confusion for new students. But you will learn this concept in lectures.)

Memory Operands

Memory is essentially a large, single dimensional array

- The address acts as the index of the array
- Addresses start at zero and go to 2^{64} -1

The value of `mem[2]` is 10

byte address

Memory addresses and contents of memory at those locations - each memory element is 1 byte

C code example A (Assuming 64-bit register/memory/data)

1 byte = 8 bits ,1 word = 32 bits, 1 doubleword = 64 bits

Memory Operand RISC-V Example

= 64 bit

A doubleword requires 8 bytes to store it
Address of subsequent doublewords thus differ by 8

The C code wants the index 8 element of the array, because each element is 64 bit (8 bytes), in RISC-V code we need to multiply index with 8.

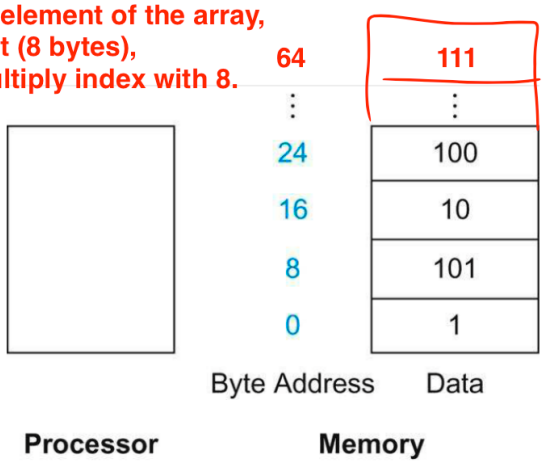
- C code:

```
A[12] = h + A[8];
```

  - h in x21, base address of A in x22

- Compiled RISC-V code:

```
ld    x9, 64(x22)     = 8 * 8
add   x9, x21, x9
sd    x9, 96(x22)     = 12 * 8
```



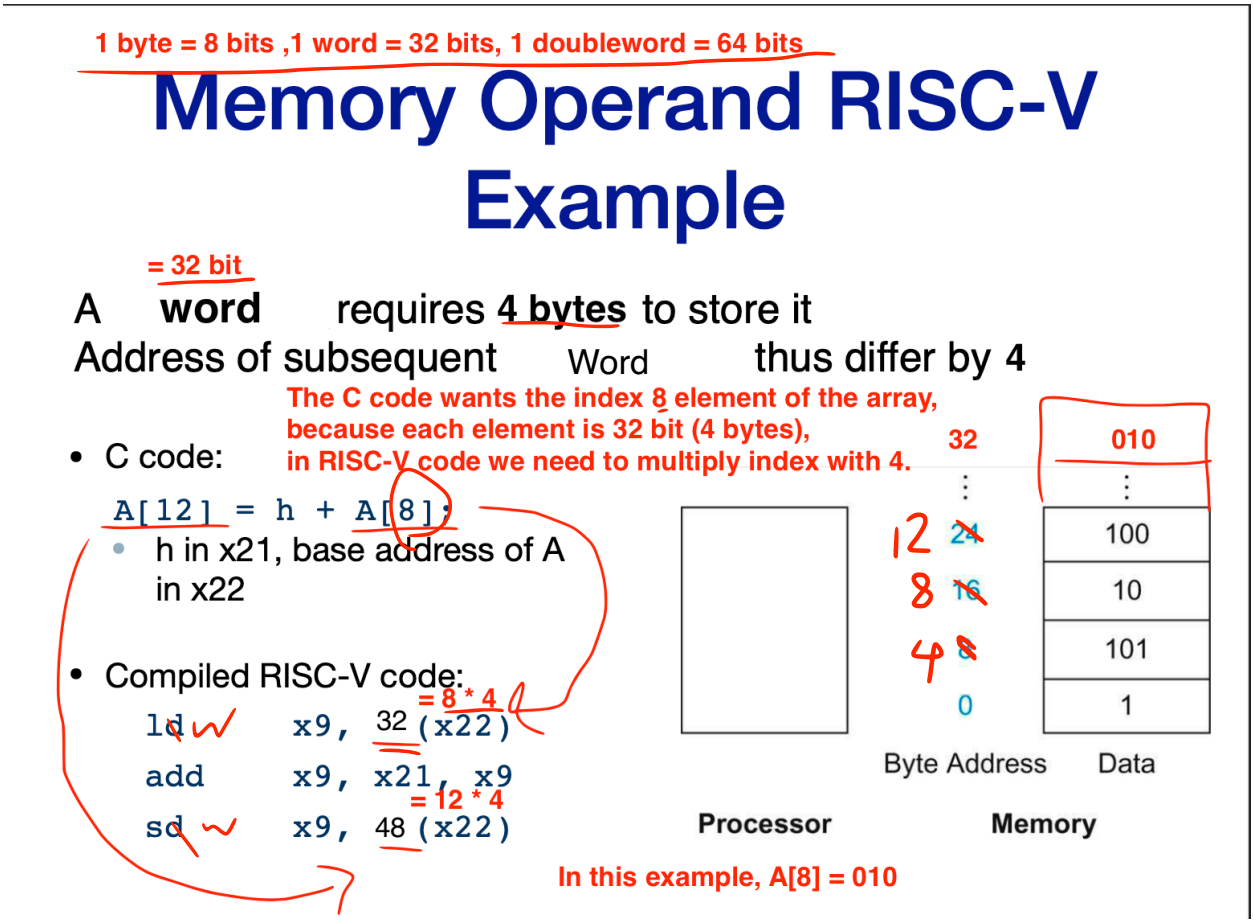
In this example, A[8] = 111

8A

13

(Note: the data value “111” in 64 byte address is arbitrary, no meaning, just for example)

C code example B (Assuming 32-bit register/memory/data)

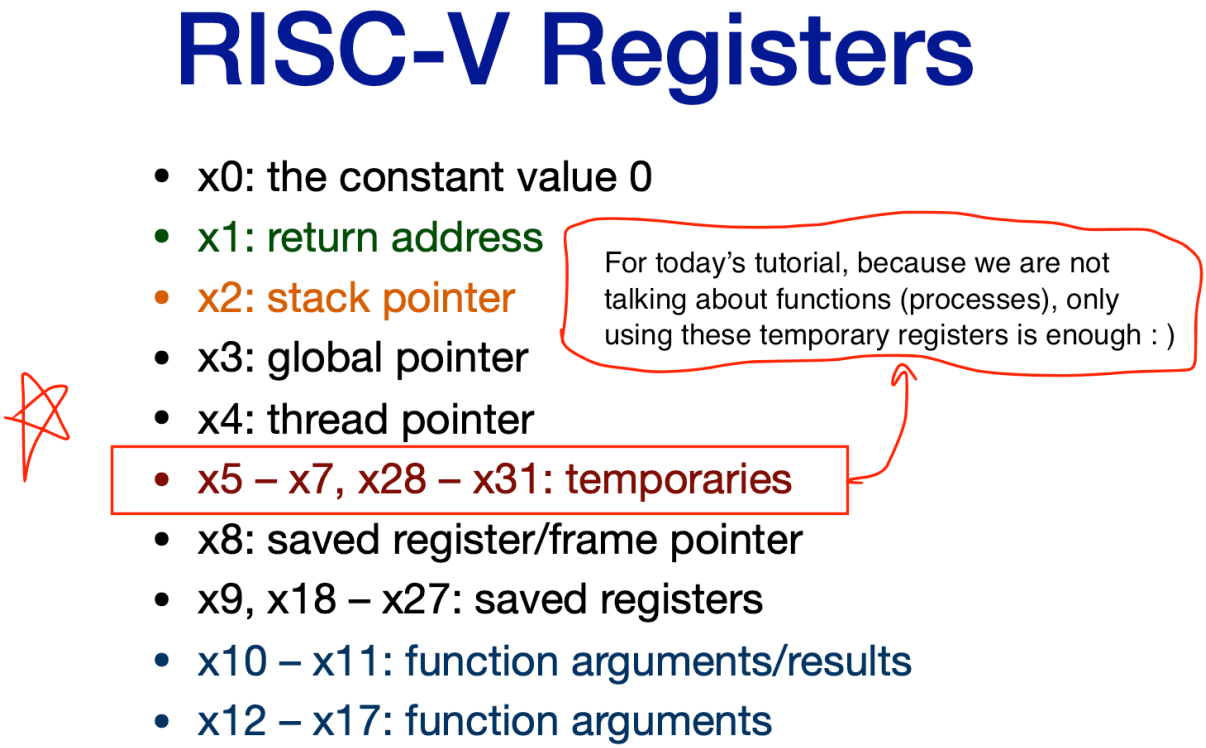


(Note: the data value “010” in 32 byte address is arbitrary, no meaning, just for example)

Register

- there are 32 registers, namely *x0-x31*, where *x0* is always zero
- to perform arithmetic operations (add, sub, shift, logical) data must always be in registers
  - the number of variables in programs is typically larger than 32, hence 'less frequently used' [or those used later] variables are 'spilled' into memory [spilling registers]
  - registers are faster and more energy efficient than memory

In another word, the register is just "faster memory". But remember all operations are on registers!!!



Arithmetic Operations

- Add and subtract, three operands, two sources and one destination.

```
add a, b, c  // store b + c in a
```

All arithmetic operations have this form.

e.g. add x10, x11, x12
means calculate x11 + x12 and store the result to x10

RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	`add x5, x6, x7`	`x5 = x6 + x7`	Three register operands; add
	Subtract	`sub x5, x6, x7`	`x5 = x6 - x7`	Three register operands; subtract
	Add immediate	`addi x5, x6, 20`	`x5 = x6 + 20`	Used to add constants
Data transfer	Load doubleword	`ld x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Doubleword from memory to register
	Store doubleword	`sd x5, 40(x6)`	`Memory[x6 + 40] = x5`	Doubleword from register to memory
	Load word	`lw x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Word from memory to register
	Load word, unsigned	`lwu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned word from memory to register
	Store word	`sw x5, 40(x6)`	`Memory[x6 + 40] = x5`	Word from register to memory
	Load halfword	`lh x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Halfword from memory to register
	Load halfword, unsigned	`lhu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned halfword from memory to register
	Store halfword	`sh x5, 40(x6)`	`Memory[x6 + 40] = x5`	Halfword from register to memory
	Load byte	`lb x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte from memory to register
	Load byte, unsigned	`lbu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte unsigned from memory to register
	Store byte	`sb x5, 40(x6)`	`Memory[x6 + 40] = x5`	Byte from register to memory
	Load reserved	`lr.d x5, (x6)`	`x5 = Memory[x6]`	Load; 1st half of atomic swap
	Store conditional	`sc.d x7, x5, (x6)`	`Memory[x6] = x5; x7 = 0/1`	Store; 2nd half of atomic swap
	Load upper immediate	`lui x5, 0x12345`	`x5 = 0x12345000`	Loads 20-bit constant shifted left 12 bits
Logical	And	`and x5, x6, x7`	`x5 = x6 & x7`	Three reg. operands; bit-by-bit AND
	Inclusive or	`or x5, x6, x8`	`x5 = x6 | x8`	Three reg. operands; bit-by-bit OR
	Exclusive or	`xor x5, x6, x9`	`x5 = x6 ^ x9`	Three reg. operands; bit-by-bit XOR
	And immediate	`andi x5, x6, 20`	`x5 = x6 & 20`	Bit-by-bit AND reg. with constant
	Inclusive or immediate	`ori x5, x6, 20`	`x5 = x6 | 20`	Bit-by-bit OR reg. with constant
	Exclusive or immediate	`xori x5, x6, 20`	`x5 = x6 ^ 20`	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	`sll x5, x6, x7`	`x5 = x6 << x7`	Shift left by register
	Shift right logical	`srl x5, x6, x7`	`x5 = x6 >> x7`	Shift right by register
	Shift right arithmetic	`sra x5, x6, x7`	`x5 = x6 >> x7`	Arithmetic shift right by register
	Shift left logical immediate	`slli x5, x6, 3`	`x5 = x6 << 3`	Shift left by immediate
	Shift right logical immediate	`srli x5, x6, 3`	`x5 = x6 >> 3`	Shift right by immediate
	Shift right arithmetic immediate	`srai x5, x6, 3`	`x5 = x6 >> 3`	Arithmetic shift right by immediate

(Note:
ld and sd instruction are typically used on 64-bit machine, while lw and sw are typically used in 32-bit machine
1 doubleword = 64 bit
1 word = 32 bit)
Trick:
Shift left a binary number by n, means multiply the number by 2^n.
Shift right a binary number by n, means divide the number by 2^n. (integer division)

Other Consideration

When compiling C code to RISC-V code, try to minimize the number of registers used and try to use the least number of instructions like a smart compiler should do :) Other topics like the how to organize code to avoid data hazard will be taught later)