Chapter 4: RISC-V

4.1 Basics of Assembly Language

Computers generally consist of binary circuits; however, it's hard to run programming circuits to run complex operations.

We need to build a circuit that can run theses sets of complex operations to describe a any behavior, which is the Central Processing Unit (CPU)

- The basic job of the CPU is to take a set of instructions and execute them in order.
    - Different CPUs implement these different sets of operations (eg. instructions) called the Instruction Set Architecture (ISA)
        - The programming language defined by ISA is known as *assembly language*
            - *ARM (cell phones), Intel x86 (i9, i7, i5, i3), IBM/Motorola PowerPC (old Macs), MIPS, RISC-V, …*

C is closer to assembly language and still more complex and automated than most

- C compilation, C memory management, C syntax, etc
- However, Assembly is almost all explicitly handled by the programmer

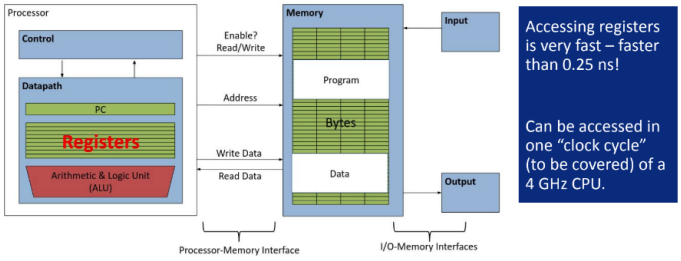
Reduced Instruction Set Computing (RISC) by IBM was to:

- Reduce the instruction set small and simple (instructions executed directly to CPU)
- Let the software do complex operations by composing simpler ones
    - Meaning simpler CPU is easier to iterate and be made faster than complex CPU (whose speed is often limited by the slowest instruction)

4.1,2 Registers (the operands of assembly operations)

The CPU can store a small amount of data, through components called *registers*. The memory location on the CPU itself

- Each register stores 32 bits of data (for a 32-bit system) or 64 bits of data (for a 64-bit system). Each register can store one *word*
- The size and number of registers is fixed but they are very fast



* Registers are labeled x0-x31

x0 is a special registers because it always hold the value 0

Unlike variables, registers have no types - this operations is what determines what the content of the register is treated as

4.2 RISC-V Structure

Instructions: each line of RISC-V code is a single instruction, which executes a simple operation on registers.

- Registers have *no types*: they are just bits.
- Operation determines type: whether treating bits as a value, memory address, etc.

The general format of a RISC-V instruction is:

- operation_name destination source1 source2
  - Ex. "add x5 x6 x7" means "Add the values stored in x6 and x7, and store the result in x5"

Addition and Subtraction of Integers in Assembly:

- Addition in Assembly:

```
add x1,x2,x3
```

Equivalent to: `a = b + c` (in C)

where `a ⇔ x1, b ⇔ x2, c ⇔ x3`

- Subtraction in Assembly

```
sub x3,x4,x5
```

Equivalent to: `d = e - f` (in C)

where `d ⇔ x3, e ⇔ x4, f ⇔ x5`

Pseudo-instructions are instructions which are translated into different instructions by the assembler

- Minimize # of instructions → minimize circuitry needed in CPU
- Ex. "mv x5 x6" is a pseudo-instruction that means "Set x5 to x6"

Immediates are numerical constants so there are special instructions for them

- Add Immediate:
  - addi x3, x4, 10
    - f = g + 10 (in C)
    - where RISC-V registers x3,x4 are associated with C variables f,g
- Similar syntax to add instructions, except last argument is a number instead register
- One particular immediate, Register Zero (x0) is hardwired to value 0:
  - Ex. add x3, x4, x0
    f=g (in C)
  - where RISC-V registers *x3, x4* are associated with C variables *f, g*

4.3 RISC-V Instruction Types

Addition/Subtraction: add, sub, addi

Bitwise: and, or, xor, andi, ori, xori

Shifts:

sll (Shift Left Logical),

srl (Shift Right Logical),

sra (Shift Right Arithmetic),
slli, srli, srai

Set Less Than: slt, sltu, slti, sltiu

Arithmetic vs Logical Shifts:

Logical Left shift: Move all bits to the left, appending zeros as needed

Logical Right shift: Move all bits to the right, prepending zeros as needed

Conclusion: To right-shift (or extend) a signed number, fill in the extra bits with the MSB to keep the same represented value. This is known as "sign-extension."

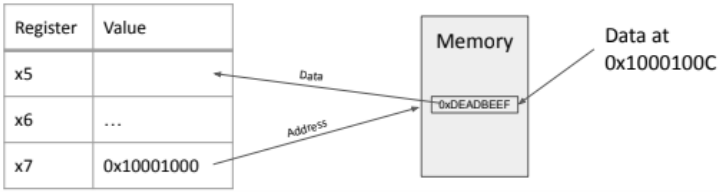
4.4 Memory Instructions

Allows data to be transferred from main memory to registers and vice versa.

Has a different syntax compared to arithmetic instructions:

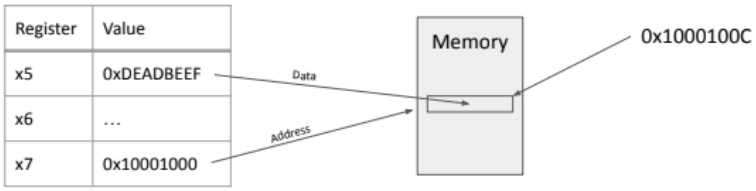
1w x5 12(x7)

"Get address in x7 and add 12. Retrieve the next word of data (4 bytes) from memory, and save the result in x5"



sw x5 12(x7)

"Interpret x7 as an address and add 12. Overwrite the 4 bytes of memory at that location with the value currently stored in x5."



RISC-V Example: Load from Memory to Register

RISC-V Example: Store from Memory to Register

4.5 Control Instructions

Control instructions instead specify a different line of code to run next.

1. Conditional jumps (branches), which jump to the specified line only if certain condition is met, and otherwise move to the next line (eg. if statements)
    a. Types of Branches:
        i. Beq: branch if the two registers have equal value
        ii. Bne: branch if non-equal

iii. Blt, bge, bltu, bgeu: Less than and “greater or equal”, along with unsigned variants.

2. Unconditional jumps, which jump to the specified line no matter what
    a. Two versions: jump to a label, jump to an address saved in a register
        i. Ex. *jal x1 Label* (Jump and Link)
            Meaning: Set x1 to PC+4 (the line of code immediately after the current line), then jump to Label
        ii. Ex. *jalr x1 x5 0* (Jump and Link Register)
            Meaning: Set x1 to PC+4, then jump to the line of code at address x5+0

We need to specify a line of RISC-V code to jump to when working w controls
Label: A human-readable identifier/location for a particular line of code.

Ex:

```
        addi x5 x0 0
        addi x6 x0 10
Loop:   add x5 x5 x6
        addi x5 x5 -1
        bne x5 x0 Loop
```

4.6 Functions

Effectively the same as labels (RISC-V makes no distinction between labels for functions and labels for loops.)

The link from a *jal* is designed specifically to allow you to return with a corresponding jr.
This link is referred to as the *"return address".*

In order for a function to "work", we have to specify:

- Which registers contain the *function arguments*
- Which register contains the *return address*
- Which register will contain the *return value*
- Which registers (if any) *can potentially be modified* by our function.

```
func_example:
  arg1 = a0                ; First argument
  arg2 = a1                ; Second argument
  saved1 = s0              ; Saved register 1
  saved2 = s1              ; Saved register 2
  return_address = ra      ; Return address register
  stack_pointer = sp       ; Stack pointer register
  next_instruction = pc    ; Program counter
  ; Rest of the function code
  Ret
```

4.6.1 Function Control Flow

- Put parameters where the function can access them (*a0-a7*):
  - Before calling a function, the caller places function parameters in registers a0-a7 so that the callee can access and use them.
- Transfer control to the function (*jump*):
  - The caller uses a jump instruction to transfer control to the callee, initiating the function call.
- Acquire the local storage resources for the function (*Increase stack*):
  - The callee allocates space on the stack to store its local variables and other necessary information. This is often done by decrementing the stack pointer (sp).
- Perform the desired task of the function:
  - The callee executes its code, using the provided parameters and stack space as needed to perform its task.
- Put the result in a0 where the calling function can access it:
  - If the function has a return value, it places the result in register a0, which the calling function can access upon return.
- Release local variables and return data to used registers so the caller can access them (*Decrease stack)*:
  - The callee cleans up its local variables and returns any necessary data to registers. This often involves incrementing the stack pointer to "pop" the stack.
- Return control to the calling function *(Jump to ra*):
  - The callee returns control to the calling function by jumping to the address stored in the return address register (ra).

Caller-Callee Convention:

- Registers sp, gp, tp, s0-s11 are preserved across a function call.
  - This means that the callee must save and restore the values of these registers to ensure they remain unchanged for the caller.
- Registers t0-t7 and a0-a7 are not preserved across a function call.
  - The callee can freely modify these registers, and the caller cannot assume their values will remain the same after the call.

Stack Frame:

- The stack frame is used to store variables and other information that need to be preserved across function calls according to the caller-callee convention.

The prologue of a function is :

- Where the necessary registers (s0-s3 in our example) are saved onto the stack to create the stack frame.
    - This typically involves decrementing the stack pointer (sp) and storing the values of the saved registers.

The epilogue of a function:

- Which occurs before the function returns, is where the saved registers are restored from the stack.
    - This ensures that the state of the callee is restored to what it was before the function call, adhering to the caller-callee convention.

4.7 Directives:

.text: This directive is used to specify that subsequent items in the assembly code belong to the text segment of memory, which typically contains the code instructions.

.data: This directive designates that subsequent items are placed in the data segment of memory, often used for static variables and data storage.

.globalsym: It declares a symbol to be global, meaning it can be referenced from other files, enabling global scope for that symbol.

.string: This directive is used to store a null-terminated string in the data memory segment.

.word: It is used to store 32-bit quantities into contiguous memory words. This is commonly used for data storage.

Instruction Formats:

- In the RISC-V architecture, instructions are represented as bit patterns and are stored in memory just like data.
- The Program Counter (PC) keeps track of the memory address of the current instruction.
- Instructions are typically 32 bits long and are divided into different fields to convey information to the processor.

Instruction Types:

R-Type (Register-register arithmetic):

- This type of instruction format is used for register-to-register arithmetic operations.
    - It includes fields for opcode, source register 1 (rs1), source register 2 (rs2), destination register (rd), funct3, and funct7.

I-Type (Register-immediate arithmetic):

- This format is used for instructions involving a register and an immediate value.
    - It includes fields for opcode, rs1, funct3, rd, and an immediate value (Imm).

S-Type (Store instructions):

- These instructions are used for storing data in memory.
    - It includes fields for opcode, rs2, rs1, funct3, and an immediate value (Imm).

B-Type (Branch instructions):

- Branch instructions are used for conditional branching.
    - This format includes fields for opcode, rs2, rs1, funct3, and an immediate value (Imm).

U-Type (20-bit upper immediate instructions):

- These instructions involve 20-bit immediate values and are often used for loading constants into registers.
    - It includes fields for opcode, rd, and an immediate value (Imm).

J-Type (Jump Instructions):

- Jump instructions are used for unconditional jumps.
    - This format includes fields for opcode, rd, and an immediate value (Imm).

Addressing Modes (PC-Relative Addressing):

- PC-relative addressing is a method used in instruction encoding for J-Type and B-Type instructions in RISC-V. It involves encoding a label as an offset from the program counter (PC).
- Because each instruction in RISC-V is one word (32 bits) long, addresses are always aligned, and the least significant bit is always 0. This means that in B-Type and J-Type instructions, the last bit of the immediate value is automatically 0 and doesn't need to be stored explicitly.

Compiling, Assembling, Linking, and Loading (CALL):

- These are the four steps involved in preparing and executing a program.

  Compiler (Compiling):
    - A compiler takes high-level programming language code as input and produces assembly language code that is specific to the machine on which the code was compiled.
    - The output of the compiler may still include pseudo-instructions in assembly code.

  Assembler (Assembling):
    - The assembler converts assembly language code into machine language code.
    - It reads and uses directives, replaces pseudo-instructions with their actual equivalents, and generates machine code (bits).
    - The output of the assembler is an object file, which includes various elements like the object file header, text segment (code), data segment (static data), relocation table, symbol table, and debugging information.

Object File Elements:

- Object File Header: Contains information about the size and position of different sections of the object file.
- Text Segment: Contains the machine code instructions.
- Data Segment: Represents the binary representation of static data from the source code.
- Relocation Table: Stores information about lines of code that need to be fixed during the linking process.
- Symbol Table: Lists global and static data labels for the program.
- Debugging Information: Contains information useful for debugging the program.

Two Passes of Assembling:

- The assembler often operates in two passes through the code.
    - In the first pass, it identifies all labels and addresses in the code, including those that may not yet be defined. Labels that are defined are stored in the relocation table.
    - In the second pass, the assembler processes the code again. When it encounters labels, it looks in the relocation table to resolve their addresses. If a label's position was defined, the assembler replaces the label with its immediate value and converts it into machine code. Otherwise, it marks the line for relocation.
- This two-pass approach helps resolve the *"forward reference problem,"* where labels are used before they are defined in the same file.

4.8 Linker

The Linker is responsible for combining all the object files produced during the compilation and assembling phases to create the final executable program.

The Linker typically operates in three main steps:

- Putting the text segments together: This involves combining the machine code instructions (text segments) from various object files into a single contiguous block of memory.
- Putting the data segments together: Similar to text segments, data segments (static data) from different object files are combined into a single data block.
- Resolving referencing issues: The Linker resolves any references to symbols or addresses that were defined in other object files. It ensures that all references are correctly linked to their absolute addresses.

The Linker knows the length of each text and data segment and can order them appropriately.

- It assumes that the first word of the program will be stored at a specific memory address (e.g., 0x10000000) and calculates the absolute address of each word from there.
- To resolve references, the Linker uses the relocation table, which specifies whether certain addresses need to be relocated or not. For example:
    - PC-Relative Addresses: These are never relocated.
    - Absolute Function Addresses: These are always relocated.
    - External Function Addresses: These are always relocated.
    - Static Data: Static data addresses are always relocated.

When the Linker encounters a label or reference, it does the following:

- Searches for the reference in symbol tables.
- Searches for the reference in libraries if it's not found in the symbol tables.
- Fills in the machine code once the absolute address is determined.

This approach is known as Static Linking because the executable doesn't change once it's linked

- In contrast, Dynamic Linking involves loading libraries during runtime, which can make programs smaller but adds runtime overhead because libraries must be searched for during execution.

4.9 Loader

The Loader is responsible for taking the executable program produced by the Linker and running it on a computer.

The Loader typically performs the following steps:

- Loads text and data segments into memory: It copies the machine code and data segments from the executable file into memory.
- Copies command-line arguments into the stack: Command-line arguments passed to the program are often copied into the program's stack for access.
- Initializes the registers: It sets up the CPU registers to their initial values as required by the program.
- Sets the Program Counter (PC) to the starting address of the program and begins execution.

The Loader essentially prepares the program for execution and hands control over to it, allowing it to run on the computer.