Syllabus

Course Description

This course introduces the concepts of computer architecture by going through multiple levels of abstraction, the numbering systems and their basic computations. It focuses on the instruction-set architecture of the MIPS machine, including MIPS assembly programming, translation between C and MIPS, and between MIPS and machine code. General topics include performance calculations, processor datapath, pipelining, instruction level parallelism, and memory hierarchy, including cache memories.

Required Textbook

Computer Organization and Design 5th Edition

Course Materials

Assembler/Simulator: MARS MIPS

Syllabus: Syllabus

Grading Criteria

1. Exams (10%, 20%, 30%): open book, 2 midterms and 1 final
2. Assignments (20%): 8, around one per week
3. Projects (20%): details TBA

Assignment grading policy:

1. Code Development (30%): compiles without errors
2. Program Execution (20%): runs successfully
3. Program Design (25%): conforms to spec
4. Documentation (15%): program comments
5. Coding Style (10%): clear and efficient

No late homework or assignments.

Day 1: Jan 19

General

Reference notes file: **Assembler.pdf** (on eLearning)

Course will be using **MIPS**.

Computer Architecture:

ISA: Instruction Set Architecture

- **RAM** (Random Access Memory) communicates with the **CPU** as well as the storage disks
- **CPU** contains **registers** and **data path**
- **Von Neumann Principle** states that instructions and data resides in RAM (invented by Alan Turing)
- The larger the size, the slower the speed

MIPS RAM Memory Layout

- Reserved
- Stack $\Longrightarrow$ Dynamic Data $\Longleftarrow$ Heap
- Static data
- Text
- Reserved

Instruction Execution

- **Data**: flows between Memory and CPU (both ways)
- **Instructions**: only flows from **Memory** to CPU
- MIPS uses 32-bit integer registers $\Longrightarrow$ 16 Double long floating point
- Loop process (infinite loop, Instruction Fetch Cycle)
  1. Fetch instruction from RAM (Program Counter)
  2. Increment PC
  3. Execute instruction (may change PC)
  4. **ONLY** terminated by interrupts from outside source

Registers

There are 32 registers, and each of them are 32 bits long. All ALU instructions have 3 addresses and each is 5 bits, for example:

```
ADD $1, $2, $3
```

There are many reserved registers so be careful when choosing them.

Types of Instructions

1. **R-Type** (ALU functions): op, rs, rt, rd (result), shamt, funct
    1. Shift, add, multiply, divide, etc.
2. **I-Type** (loads and stores): op, rs, rt (result), imm
    1. Must load data before operating on them
    2. Store result back in store operation
    3. Around 20% of operations are I-Type
3. **J-Type** (unconditional branch): op, addr
    1. Jump somewhere

Clock Cycle

All RISC instructions take one **clock cycle** to execute.

Execution example:

(Note: `Mem`=read from memory, `WB`=Write Back to memory)

```
IF -> ID -> IX -> Mem -> WB (I300)
      IF -> ID -> IX -> Mem -> WB (I301)
            IF -> ID -> IX -> Mem -> WB (I302)
```

Split instructions into smaller chunks so each can execute in one clock cycle. However, there are issues relating to dependencies between different instructions which causes delays. Goal is to get maximum throughput through the MIPS pipeline.

Coming up next: performance evaluation and some Law.

Day 2: Jan 21

Fetch-Execute Loop

```
for(;;) {
    Instruction Fetch
    PC += d // optional
    Instuction Execute
}
```

The **PC** (Program Counter) keeps track of what instruction to fetch.

Variables, Constants, etc.

Variables is initially stored in **RAM** when the program is initially loaded. Not the entire program is loaded, however, since programs could be very large. Other bits of the program will be loaded as the program goes on in a process called **Virtual Memory**. Once the initial **working set** of the program is loaded in RAM, we can begin executing instructions.

Instruction Sets

The earlier computers in the 1940's are very basic and could only perform the very fundamental operations. Thus, programmers must write their own **subroutines** (functions) to provide additional functionalities.

MIPS has support for add, subtract, multiply, divide, bit-wise AND, OR, XOR, and shifting.

MIPS is a **Reduced Instruction Set Computer** (RISC), which has far fewer functions compared to more complex processors.

Instruction Set Architecture

Memory Representation
Memory addresses are stored as **32-bit** values given in hex, and they represent the *locations* of the actual data values.

Some computers have stack-based processors which supports recursive calls and general function calls. For example, adding two numbers could be represented as follows:

```
PUSH Ay
PUSH Az
ADD
POP Ax
```

The **Stack ISA** (Instruction Set Architecture) is good because it does not need named registers to store values. However, the downside is that the stack has a limited size and functions that access arrays and process strings are much more difficult to implement.

Three-Address ISA allows for instructions that can have an operation and three value addresses. For example, add can be done just as `Add Ax, Ay, Az`.

Two Address Machines is another alternative, but most 60s computers provided just one register called an **Accumulator**. For example, adding would look like this:

```
Clear
Add Ay
Add Az
Store Ax
```

Since there was only one register available, there was no need to name it.

MIPS Load-Store ISA

MIPS has 32 named registers, and it provides operations for *loading* and *storing* data, which is separate from actual arithmetic operations. Adding looks like this:

```
LW  $t6  4($a0)    # load value from $a0 to register $t6
LW  $t7  8($a0)    # load value for the next 4 bytes to register $t7
ADD $t5  $t6, $t7  # add the first two values and store in $t5
SW  $t5  0($a0)    # stored the sum into the initial location
```

Note that in this example, the first 4 bytes are for the result, the next 4 (starting from 4) for the first number, and the next four (starting from 8) is for the second number.

A key distinction is the memory *address* vs actual *data values*. We use the memory addresses to tell the program where to find the data, but the memory addresses are *not* the actual value.

MIPS has support for different *address modes*, which has different meanings and functions. One example is adding with register mode versus adding in immediate mode (see notes for more details). Using immediate instructions allow you to directly specify what to add to the previous number without directing it to a memory address first.

To evaluate an ISA, we usually have a **benchmark set** that can be used to determine the length of the program (which affects the size of RAM we need) as well as the execution time.

Load and Store RISC

Characteristics of RISC machines:

1. Fixed instruction size (32 bits)
2. Fixed instruction execution time for all integer instructions (each instruction takes 1 clock cycle)
3. High clock rate from simplicity
4. Load and store are the only ways to move data between RAM and processor
5. All instructions contain three register addresses

There are many decisions made while designing these ISAs. We'll be going over the similarities and differences between various ISAs as well as why certain things are done a certain way.

Metrics

Computers are often abstracted for most end-users. For example, students might be using the computer for electronic files and interacting with it through high-level languages like Java and C++. Applications software are exposed to the end-users for simple user interaction, then there is the systems software layer which functions as a connection between the applications and the hardware layer.

Executable files on a computer are loaded into RAM by a *loader* and executed by the processor. The processor handles the instructions through different implementations, and the next level is the hardware with billions of transistors with boolean gates.

The compiler's job is to turn high-level language into assembly language, and the assembler takes the assembly language program into binary machine code.

Different computers have different instructions and other functions, so we must design a suitable performance metric that can be used to evaluate the efficiency. CPU time is calculated as follows:

```
CPU Time = IC * CPI / (Clock Rate)
```

Where IC = instructions per program (instruction count) and CPI = cycles/instruction.

Day 3: 1/26

CPU Time

$$\textbf{CPU Time} = \frac{Seconds}{Program} = \frac{Instructions}{Program} \cdot \frac{Cycles}{Instructions} \cdot \frac{Seconds}{Cycle}$$

$$\textbf{CPU Time} = IC \cdot CPI \cdot \frac{1}{ClockRate}$$

IC is instruction count and **CPI** is cycles per instruction.

Table 1: Example of Calculating CPI

Op-Code	Frequency	Cycles	CPI (i)	Time
ALU	50%	1	0.5	33%
Load	20%	2	0.4	27%
Store	10%	2	0.2	13%
Branch	20%	2	0.4	27%

The overall CPI is the sum of all CPI entries:

$$CPI = 0.5 + 0.4 + 0.2 + 0.4 = 1.5$$

In this case, we can see that we spent half the time on ALU, so we should **make the common case fast**. This is because architects might try to optimize some system, but later realize that there is little speed up. To prevent that, we should measure the usage of those operations (ALU in this case) before optimizing.

Since we always have trade-offs when making decisions on what to improve, we should always prioritize the more frequent events (ALU for example) over the less frequent cases (Store for example). The rule of thumb is to only improve if the benefit is **over 1%**.

Improvement Examples

Let's say that division was improved by a factor of 2 for its runtime, then the speedup is

$$Speedup = \frac{1}{(1 - 0.03) + \frac{0.03}{2}} = 1.015$$

Since 1.015 means a 1.5% improvement, it is a reasonable improvement to consider.

Note: using arithmetic mean to compare the runtimes of different programs on different systems is not a good idea. It could remove drastic differences that could be seen by directly observing the data points (for example, 1 very high run time and 9 very low run time).

The better (and usual) approach to **normalizing** the runtimes is to use a *reference machine*, and calculate the other runtimes by multiplying by the reciprocal of the reference runtime.

Geometric Mean is a metric that does not require a reference machine. It is accepted by the results does not relate easily to well-known metrics (MIPS, MFLOPS, IPC, CPI).

Another metric is the arithmetic mean of **IPC** (instructions per cycle). The opposite is using the sum of **CPI** (cycles per instruction). Since these two are reciprocals, the resulting metrics are also reciprocals.

Note: For notes on signed/unsigned numbers, see `Numbers.pdf` on eLearning.