CS2100 Computer Organisation
AY24/25, Y2S1
Notes

Sim Ray En Ryan

December 3, 2025

Contents

1 Course Administration

1.1 Assessment

- Tutorial Attendance (5
- Canvas Quizzes (3
- 3 Assignments (12
- Labs (10
- Midterms (17.75
- Finals (46.25

2 Introduction

2.1 Programming Language

- A formal language that specifies a set of instructions for a computer to implement specific algorithms to solve problems.
- **High Level**: Python, Java, C (High Abstraction, Productivity, Portability)
- **Low Level**: MIPS (Low Abstraction, Assembly)

2.2 Language Types and Execution

- **Scripting (JS, PHP, Python)**: Interpreted, Slower.
- **Programming (C, Java)**: Compiled, Faster.
- Abstraction Hierarchy: High Level → Assembly (ASM) → Binary (Machine Code).

2.3 Computer Components

- **Instruction Set Architecture (ISA)**: The interface between hardware and low-level software.
- **Core Components**: Processor (CPU), Memory, I/O.
- **Processor Design**: Datapath and Control Design → Digital Logic Design.
- **Computer Model**: Input → CPU (Control Unit, ALU) → Memory → Output.
- Example C Code (Sum of Squares):

```
int main() {
int i, a = 0;
```

```
for(i=1; i\<=10; i++) {
a = a + i\*i;
}
}
```

- Example MIPS Assembly Code:

```
main: addi $t1, $zero, 100
add $t3, $zero, $zero
Loop: add $t2, $zero, $zero
add $t4, $t1, $zero
Mul:  add $t2, $t2, $t1
addi $t4, $t4, -1
bne $t4, $zero, Mul
add $t3, $t3, $t2
addi $t1, $t1, -1
bne $t1, $zero, Loop
```

3 C Programming

3.1 Programming Process

- Edit → Compile → Execute.

3.2 Program Structure

3.2.1 Preprocessor Directives

- **Macro expansions (ALL CAPS)**: t#define PI 3.142 defines constants. The preprocessor replaces all instances of tPI with t3.142 before compiling.
- Conditional compilation.

3.2.2 Reserved Words

- tint, tvoid, tdouble, treturn.

3.2.3 Variables

- Declaration formats:
  - Partial initialization is possible.
- An uninitialized variable will contain a random value.

- **Rules for variable names**: Cannot begin with digits, only contain alphanumeric and t_, cannot be reserved keywords, and avoid standard identifiers like printf
- Assignment is **right-associative**: ta = b = c = 9 is evaluated as ta = (b = (c = 9)). Assignment returns the assigned value (e.g., 9) as a side effect.

3.2.4 Execution Structures and Operators

- **Selection Structures**: tswitch, tif, telse, conditional operators.
- **Repetition Structures**: tfor, twhile, tdo-while. tbreak and tcontinue are flow control keywords.
- **Increment/Decrement Operators**:
  - tx++ (Post-increment): Returns the original value of tx, then increments x by 1.
  - t++x (Pre-increment): Increments x by 1, then returns the new value of x.
- **Order of Operations**: Modulus (
- **Binary operators** are generally **left-associative**, while **unary operators** are **right-associative**.

3.3 Von Neumann Architecture

- Components: CPU (Registers, Control Unit with Instruction Register and Program Counter, ALU), Memory, I/O.
- **Key Feature**: Stores both program (instructions) and data in the same memory (RAM).

3.4 Data Types and Variables

- Variables have a **Name** (Identifier), **Data Type**, **Value**, and **Address**.
- **Primary Data Types (in bytes)**:
  - tint (4 bytes): Range $\approx -2^{31}$ to $2^{31} - 1$.
  - tfloat (4 bytes)
  - tdouble (8 bytes)
  - tchar (1 byte): **Not a string**. Single characters use quotes (t'), strings (char arrays) use double quotes (t"").
- **C is Strongly Typed**: Every variable must be declared with a data type.

3.5 Main Function and I/O

- tint main(void) : Returns 0 if successful (compilers often add treturn 0; if forgotten).
- **Input (tscanf)**: tscanf(input param, address[es]). All user inputs are initially read as strings, then converted. Returns the number of items that have been successfully assigned.

4 Data Representation and Number Systems

4.1 Data Representation

- Data is fundamentally a sequence of **bits** (binary digits).
- N bits can represent 2^N values.
- $\lceil \log_2 M \rceil$ bits are required to represent M values.
- **Number Systems**: Decimal, Binary, Octal, Hexadecimal, and Radix R systems are all **weighted-positional** number systems.

4.2 Number System Conventions (in C and QTSpim)

- In **C**: t032 is Octal (32_8), t0x32 is Hexadecimal (32_{16}).
- In **QTSpim**: t0x100 is Hexadecimal (100_{16}).

4.3 Conversion Between Bases

- **Integer Part (Base M to N)**: Successive division by N. Read remainders from bottom up. Example: $11_{10} \rightarrow 1011_2$.
- **Fractional Part (Base M to N)**: Successive multiplication by N. Read the integer part from top down. Example: $0.3125_{10} \rightarrow 0.0101_2$.
- **Binary to Octal**: Group binary digits in sets of 3, starting from the floating point.
- **Binary to Hexadecimal**: Group binary digits in sets of 4, starting from the floating point.

4.4 Character Encoding

- **ASCII**: Uses **7 Bits**. The 8th bit is often used as a "checksum" parity bit for error checking.

4.5 Representing Negative Numbers

4.5.1 Signed Magnitude

- The first bit is the **sign bit**: 0 for positive (+), 1 for negative (-).
- **Range** (for n bits): $-(2^{n-1}-1)$ to $2^{n-1}-1$.
- **Disadvantages**: Arithmetic is complex. Has **two representations for zero** ($+0$ and -0).

4.5.2 One's Complement

- **Negation**: Flip all bits (a XOR with all 1s). Trivial to implement in hardware.
- **Range** (for n bits): $-(2^{n-1}-1)$ to $2^{n-1}-1$.
- **Disadvantage**: Still has **two representations for zero**.

4.5.3 Two's Complement

- **Negation**: Flip all bits, then add 1 (equivalent to $2^n - x$).
- **Range** (for n bits): -2^{n-1} to $2^{n-1}-1$.
- **Advantages**: **Only has one zero**. Has a larger range (by 1 magnitude) than one's complement. Simplifies binary arithmetic.

4.5.4 Two's Complement Addition/Subtraction

- Perform binary addition of the two's complement numbers.
- **Ignore the carry out of the MSB**.
- Check for **overflow** (when the sign of the result is incorrect).

4.6 Excess Representation

- Used for **comparison**, particularly in floating-point exponents.
- A number x is represented by the binary code for $x + \text{Excess } n$.
- **Excess** n means n is added to each number to reach its binary representation. Example: Excess 4 in 3-bit: -4 is 000, -3 is 001, etc.
- Common biases are 2^{n-1} or $2^{n-1}-1$.

4.7 Real Numbers

4.7.1 Fixed Point Representation

- Bits allocated for the integer and fractional parts are fixed.
- Provides a **limited range**.

4.7.2 Floating Point (IEEE 754)

- Number is represented as **Sign** $\times 2^{\text{Exponent}} \times$ Mantissa.
- The sign bit only applies to the mantissa. Base is assumed to be 2.
- **Single Precision (32 Bits)**: 1 Bit Sign, 8 Bits Exponent with a bias of 127, 23 Bits Mantissa.
- **Double Precision (64 Bits)**: 1 Bit Sign, 11 Bits Exponent with a bias of 1023, 52 Bits Mantissa.
- **Normalized Form**: The implicit leading 1 in the scientific notation ($1.xxx$) is not stored in the mantissa.

5 Pointers and Functions (C)

5.1 Pointers

- Pointers store the **address** of a variable.
- **Address-of operator**: $t\&a$ gives the memory address of variable ta.
- **Pointer declaration**: tint a_ptr; or tint a, a_ptr.
- **Initialization**: tint a = 123; int a_ptr = &a;
- Addresses are printed in **hexadecimal** using the format specifier t
- Addresses are generally different for every run of a program due to **Address Space Layout Randomization** (ASLR) for security.
- **Uses**: "Return" two or more variables from a function, and working with arrays.

5.2 Functions

- Functions should have **prototypes** (declarations) before the tmain function.
- **Pass by Value**: Default C mechanism where a copy of the argument is passed to the function.

- **Scope Rule**: Local parameters and variables are only accessible in the function they are declared in (**Automatic Variables**).
- **Static Variables**: Exist in memory even after the function finishes execution.

6 Arrays, Strings and Structures (C)

6.1 Array

- A collection of data elements of **the same type** (e.g., tint c[30]).

6.2 Strings

- An array of characters followed by a null character, t\0.
- The null terminator is **essential**.
- It is **automatically added** when initializing with double quotes: tchar fruit_name[] = "apple";.
- Manual initialization requires t\0: tchar fruit_name[] = 'a','p','p','l','e','\0';.
- **Input**:
  - tfgets(str, size, stdin): Reads tsize - 1 characters or until t\n. May need to replace t\n with t\0.
  - tscanf("
- **Output**:
  - tputs(str): Prints the string and automatically terminates with a newline.
  - tprintf("

6.3 Struct (Structure)

- A user-defined aggregate data type that allows grouping variables of **different data types**.
- A structure is a **definition of a type**, not a variable.
- **Declaration (using ttypedef)**:

```
typedef struct {
int length, width, height;
} box\_t;
```

- **Initialization**: tbox_t box1 = 1, 2, 3;.

- **Accessing Members**:
  - Using the **dot operator** (t.): tbox1.length = 1;.
  - Using the **arrow operator** (t-) for pointers to structs.
- When a struct is passed to a function, a **separate copy** is created (**pass by value**).

7 MIPS Assembly Language I

7.1 Instruction Set Architecture (ISA)

- An **abstraction** defining the interface between the hardware and low-level software.
- It includes all components required for machine code to function.

7.2 Machine Code vs. Assembly (ASM)

- **Machine Code**: Binary instructions executed directly by the hardware.
- **ASM**: Symbolic, human-readable instructions that map to machine code.

7.3 Data and Instructions

- Code and data reside in **Memory**.
- During execution, they move to the **CPU** (ALU).
- Data is temporarily stored in small, fast **Registers**.
- **Instructions are used to**:
  1. **Load** data from memory to registers.
  2. **Store** data from registers to memory.
  3. Perform calculations and control flow.

7.4 Registers

- **Fast** but **small** and have **no explicit data type**.
- In MIPS, there are **32 registers** (t0tot31).
- Registers tt0tott7 are typically used for immediate/temporary results.

7.5 MIPS Instructions and Syntax

- **Syntax**: top destination, source 1, source 2 (e.g., tadd $s0$,s1, $s2$).**Arithmetic Operations** :

- - tadd, tsub (Register-Register).
  - taddi (Add Immediate): Source 2 is a **constant** (immediate value) ranging from -2^{15} to $2^{15} - 1$.

Pseudo-Instruction: tmove $s0$,s1 translates to tadd $s0$,s1, *zero*.

7.6 Bitwise Operations

- **Bitshifts**: tsll (Shift Left Logical), tsrl (Shift Right Logical).
  - Shifting is often **faster** than multiplication and division.
  - Uses tshamt (shift amount), which is 5 bits.
- **Logical Operations**:
  - tand, tandi (**Masking**).
  - tor, tori (**Forcing** certain bits to 1).
  - tnor (Bitwise NOT is achieved using tnor).
  - txor, txori.

8 MIPS Assembly Language II (More Instructions)

8.1 Memory Access

- A **Word** is 4 bytes (32 bits) in MIPS.
- For loading and storing words, memory addresses must be at **multiples of 4** (word-aligned).
- **Load/Store Instructions**:
  - **Word**: tlw (Load Word), tsw (Store Word).
  - **Halfword** (16 bits): tlhw, tshw. tulw, tusw for unaligned access.
  - **Byte** (8 bits): tlb (Load Byte), tsb (Store Byte).

8.2 Decisions and Control Flow

- **Conditional Branches**:

  - tbne $t0$,t1, label (Branch if not equal).
  - tbeq $t0$,t1, label (Branch if equal).
- **Unconditional Jump**:
  - tj label.
- **Inequalities (Set Less Than)**:
  - tslt $t0$,s1, $s2$($Sett$t0 to 1 if $t\$s1 < t\$s2$, else 0).
  - Used with a branch: tbne $t0$,zero, label.

9 MIPS Instruction Formats and Encoding

9.1 General Encoding Principles

- All MIPS instructions have a **fixed length of 32 bits**.
- Encoding is designed to be as **regular** as possible.

9.2 R-Format (Register)

- Used for instructions with **2 source registers** and **1 destination register**.
- **Fields (32 bits total)**:

Field	**Opcode**	**rs**	**rt**	**rd**	**shamt**	**funct** height**Bits**
6	5	5	5	5	6 height	

- **Opcode**: Typically t000000 for R-Format.
- **rd**: Destination register.
- **shamt**: Shift amount.
- **funct**: Combines with opcode to fully specify the instruction.

9.3 I-Format (Immediate)

- Used for instructions with **1 source register**, **1 destination register**, and an **immediate value** (e.g., taddi, tlw, tsw, branches).
- **Fields (32 bits total)**:

Field	**Opcode**	**rs**	**rt**	**Immediate** height**Bits**
6	5	5	16 height	

- **Immediate** (16 bits): A signed two's complement integer, ranging from -2^{15} to $2^{15} - 1$.

9.4 Branch Encoding (I-Format)

- The 16-bit timmediate field is used as a **word offset**.
- **PC Calculation**:
  - If Branch is **NOT taken**: PC = PC + 4.
  - If Branch is **taken**: PC = (PC + 4) + (immediate × 4).

9.5 J-Format (Jump)

- Used for **unconditional jumps**.
- **Fields (32 bits total)**:

Field	**Opcode**	**Target Address** height**Bits**
6	26 height	

- The 26-bit address is implicitly multiplied by 4 (word-aligned), yielding 28 address bits.

10 Instruction Set Architecture (ISA)

10.1 CISC vs. RISC

- **CISC** (e.g., x86-32): Complex instructions, smaller program size, complex hardware.
- **RISC** (e.g., MIPS, ARM): Simple instructions (fixed length), easier hardware implementation, burden on software.

10.2 Operand Storage and ISAs

- **Stack Architecture**: Operands implicitly on top of stack.
- **Accumulator Architecture**: One operand implicitly in a special register.
- **General Purpose Register (GPR) Architecture** (Most used):
  - **Register-Memory**: One operand in memory.
  - **Register-Register (Load-Store)** (RISC): Operations only on registers; Load-/Store access memory.

10.3 Memory and Addressing Mode

- **Endianness**:
  - **Big Endian (MIPS)**: MSB stored at the lowest address.
  - **Little Endian**: LSB stored at the lowest address.
- **Addressing Modes**: Register, Immediate, Displacement, etc.

10.4 Amdahl's Law

- Performance improvement gained by optimizing a part is limited by the fraction of time that the improved part is actually used.

11 The Processor: Datapath

11.1 Processor Functions

- **Datapath**: Processes data (ALU, memory ops).
- **Control**: Generates control signals.

11.2 Instruction Execution Cycle

11.2.1 Fetch Stage (IF)

- Get instruction from **Instruction Memory** using **PC Register**.
- Update PC: PC $\rightarrow$ PC + 4.

11.2.2 Decode and Operand Fetch Stage (ID)

- Decode instruction (opcode). Fetch operands from **Register File**.
- **RegDst**: Selects destination register (*trd* or *trt*) using a MUX.
- **ALUSrc**: Selects 2nd ALU operand (register data or sign-extended immediate) using a MUX.

11.2.3 Execute Stage (EX)

- Perform operation in **ALU**.
- Calculate branch target: Target = (PC + 4) + (immediate $\times$ 4).
- **ALUControl** (4-bit signal) specifies the operation.
- **isZero** flag is output (for branching).

11.2.4 Memory Access Stage (MEM)

- Access **Data Memory** (only for tlw and tsw).
- Controlled by **MemRead** and **MemWrite**.

11.2.5 Result Write Back Stage (WB)

- Write result to **Register File**.
- **MemToReg**: Selects data source (ALU result or Memory Data) using a MUX.

12 The Processor: Control

12.1 Control Signals

- Generated by the **Control Unit** (Combinational Circuit) based on Opcode/Funct code.
- Key Signals: **RegDst**, **RegWrite**, **ALUSrc**, **ALUControl**, **MemRead/MemWrite**, **MemToReg**, **PCSrc**.
- **PCSrc** is Branch AND isZero.

12.2 ALUControl Signal Decoding

- Uses an intermediate 2-bit signal **ALUOp** (00 for *tlw*/*tsw*, 01 for *tbeq*, 10 for R-Format).
- ALUOp combines with the **Funct** field (for R-Format) to determine the 4-bit ALU-Control.

13 Boolean Algebra

13.1 Digital Logic and Circuits

- **Combinational Circuits**: Output depends only on input (e.g., Gates, MUXes).
- **Sequential Circuits**: Output depends on input and state (e.g., Registers, Memory).

13.2 Boolean Algebra Laws

- Precedence: NOT ($\neg$), then AND ($\cdot$), then OR ($+$).
- Identity, Inverse, Commutative, Associative, Distributive, Duality.

- Theorems: Idempotency ($X + X = X$), Absorption 1 ($X + X \cdot Y = X$), DeMorgan's ($(X + Y)' = X' \cdot Y'$).

13.3 Canonical Forms

- **Minterms (m)**: Product term of all literals. Canonical Sum of Products ($\sum m$).
- **Maxterms (M)**: Sum term of all literals. Canonical Product of Sums ($\prod M$).

14 Logic Circuits

14.1 Complete Sets

- **AND, OR, NOT**.
- **NAND** and **NOR** are each self-sufficient.

14.2 Implementation Forms

- **SOP**: 2-level AND-OR or 2-level NAND.
- **POS**: 2-level OR-AND or 2-level NOR.

14.3 Programmable Logic

- **PLA**: Configurable AND plane $\rightarrow$ configurable OR plane (SOP).
- **ROM**: Fully decoded mapping (can implement any function).

15 Simplification (Karnaugh Maps)

15.1 Techniques

- Algebraic, **Karnaugh Maps** (K-Maps, graphical, max 6 variables), Quine-McCluskey (tabular, automation).

15.2 K-Map Minimization

- Use Gray Code labels. Group adjacent 1s in powers of 2.
- **Prime Implicants (PIs)**: Largest possible groups.
- **Essential Prime Implicants (EPIs)**: PIs covering a minterm exclusively. Select EPIs first.
- **Don't Care Conditions (X/d)**: Can be 1 or 0 to aid simplification.

16 Combinational Circuits

16.1 Design Methods

- Gate Level (SSI) or Functional Blocks (MSI).

16.2 MSI Components

- **Decoder**: n inputs $\rightarrow 2^n$ outputs (minterms). Same as DMUX with Enable.
- **Encoder**: 2^n inputs $\rightarrow n$ outputs. Priority Encoder handles multiple inputs.
- **Demultiplexer (DMUX)**: 1 input $\rightarrow 2^S$ outputs.
- **Multiplexer (MUX)**: 2^n inputs $\rightarrow$ 1 output.

17 Sequential Logic

17.1 Memory Elements

- **Latches**: Pulse Driven (output follows input while clock/enable is high).
- **Flip-Flops**: Edge Driven (output changes only on clock edge).

17.2 Latches and Flip-Flops

- **S-R Latch**: Set/Reset. $S = 1, R = 1$ is Invalid.
- **D Latch**: Single data input D. $Q(t + 1) = D$ when $EN = 1$.
- **J-K Flip-Flop**: $J = 1, K = 1$ is Toggle.
- **T Flip-Flop**: Single input T. $T = 1$ is Toggle.

17.3 System Memory

- **Memory Hierarchy**: Registers $\rightarrow$ Cache $\rightarrow$ Main Memory $\rightarrow$ Disk Storage.
- **SRAM** (Flip-flops, faster, volatile).
- **DRAM** (Capacitors, denser, cheaper, must be refreshed).

18 Pipelining I

18.1 Concept

- **Increases Throughput**, doesn't decrease Latency.

- **5 Stages**: IF $\rightarrow$ ID $\rightarrow$ EX $\rightarrow$ MEM $\rightarrow$ WB.

18.2 Pipelined Datapath

- Uses **Pipeline Registers** (e.g., IF/ID, ID/EX) to store state between stages.

18.3 Performance

- Speedup $\approx$ Number of stages (N), ideally.

19 Pipelining II (Hazards)

19.1 Structural Hazards

- Conflict over resources. Solved by separating Instruction/Data Memory or splitting Register File access (Write in 1st half, Read in 2nd half).

19.2 Data Hazards (RAW)

- Read After Write conflict.
- Solved by **Forwarding** (bypassing result directly from pipeline register).
- **Load-Use Hazard** requires a **Stall** (1 clock cycle delay).

19.3 Control Hazards

- Branch decision is late (MEM stage in basic design).
- Mitigated by **Early Branch Resolution** (comparator in ID stage), **Branch Prediction** (assume not taken), and **Delayed Branching** (moving useful instruction into the Delay Slot).

20 Direct Mapped Cache

20.1 Locality

- **Temporal**: Item referenced again soon.
- **Spatial**: Nearby items referenced soon.

20.2 Performance Metrics

- Avg Access Time = (Hit Rate $\times$ Hit Time) + (Miss Rate $\times$ Miss Penalty).

20.3 Direct Mapping

- Address breakdown: **Tag** | **Cache Index** | **Byte Offset**.
- **Cache Index** determines the single possible location.
- Hit requires Valid bit=1 AND Tag match.

20.4 Cache Misses

- **Compulsory/Cold Start**, **Conflict/Collision**, **Capacity**.

20.5 Writing Policies

- **Write Through**: Write to cache and memory.
- **Write Back**: Write only to cache; update memory only on replacement if **Dirty Bit** is set.

21 Set and Fully Associative Cache

21.1 Set Associative Cache

- Memory block maps to a **Set Index**, but can be in any of the N blocks (ways) in that set.
- Reduces Conflict Misses. Requires simultaneous tag comparison within the set.

21.2 Fully Associative Cache

- Block can be stored anywhere. Address has only **Tag** and **Byte Offset**.
- All tags must be simultaneously compared (expensive). Used for small caches.

21.3 Block Replacement Policy

- **Least Recently Used (LRU)** is typically optimal but complex to implement.