CS2115 Review

2. Number Systems

IEEE 754 Floating Point

IEEE 754 single precision floating point numbers are:

- 32 bits long
- 1 sign bit (MSB), 0 for positive and 0, 1 for negative
- 8 (biased) exponent bits e, bias = $2^{8-1} - 1 = 127$, true exponent $E = e - 127$
- 23 (normalized) significand bits m, true mantissa $M = 1.m$

3. Circuits

- Combinational circuits: output depends only on current input
- Sequential circuits: output depends on current input and previous state

Full Adder

Truth table:

A	B	Cin	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

- $S = A \oplus B \oplus Cin$
- $Cout = (A \wedge B) \vee (A \wedge Cin) \vee (B \wedge Cin)$

Circuit:

- $S = (A \oplus B) \oplus Cin$
- $Cout = (A \wedge B) \vee ((A \oplus B) \wedge Cin)$

Carry Lookahead Adder

- $C_{i+1} = G_i + P_i C_i$
- Generator $G_i = A_i B_i$ - decides if new carry is generated
- Propagator $P_i = A_i \oplus B_i$ - decides if previous carry is propagated

4-bit adder:

- Input: $A_3 A_2 A_1 A_0, B_3 B_2 B_1 B_0$
- $C_0 = 0$
- $C_1 = G_0 + P_0 C_0$
- $C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_0$
- $C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$
- $C_4 = G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0$
- Output: $C_4 S_3 S_2 S_1 S_0$
- Output: $G^I = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0$, $P^I = P_3 P_2 P_1 P_0$

16-bit adder:

- Input: $A_{15} A_{14} \ldots A_1 A_0, B_{15} B_{14} \ldots B_1 B_0$
- $C_4 = G^I + P^I C_0$
- $C_8 = G^{II} + P^{II} C_4 = G^{II} + P^{II} G^I + P^{II} P^I C_0$
- $C_{12} = G^{III} + P^{III} C_8 = \cdots$
- $C_{out} = G^{IV} + P^{IV} C_{12} = \cdots$
- Output: $C_{out} S_{15} S_{14} \ldots S_1 S_0$

SR Latch

NOR gate SR latch, active high

True table:

S	R	Q	Q'	Remarks
0	0	1 0	0 1	Keep
0	1	0	1	Reset
1	0	1	0	Set
1	1	X	X	Invalid

NAND gate SR latch, active low

True table:

S	R	Q	Q'	Remarks
0	0	X	X	Invalid
0	1	1	0	Set
1	0	0	1	Reset

S	R	Q	Q'	Remarks
1	1	0 1	1 0	Keep

Gated SR Latch (Continuous)

Only respond to set/reset when enable is 1

- AND gate + NOR gate SR latch, active high
- NAND gate + NAND gate SR latch, active high

Gated D Latch (Continuous)

> Based on NAND gate + NAND gate SR latch, active high

Use `D` as set input, `D'` as reset input

Clocked D Latch (Instantaneous)

Use a rising edge detector to generate clock signal

Compared to Gated D Latch, Clocked D Latch is only enabled at the start of every clock cycle

D Flip-Flop

Connecting a Gated D Latch and a Gated SR Latch

Use `CLK` as the enable input of master Gated D Latch, use `CLK'` as the enable input of slave Gated SR Latch

The D Flip-flop accepts input `D` when `CLK` is high (Continuous), and outputs the value of `Q` when `CLK` turns low (Instantaneous)

4. Assembly

- Instruction Set Architecture (ISA): the interface between hardware and software
  - Transform assembly instructions into machine code
  - Access memory, registers, I/O devices
  - Provide a list of instructions to the programmer
- Reduced Instruction Set Computer (RISC): fixed length (1-word) instructions, store operands in registers
- Complex Instruction Set Computer (CISC): variable length instructions, store operands in memory

Control Unit

1. Load instruction from memory
2. Decode instruction
3. Fetch operands from registers
4. Execute instruction
5. Store result in register

MIPS

MIPS is a RISC ISA. It stores operands in registers.

The register file only contains 32 32-bit registers. So transfer data between memory and registers is required.

Memory is a large 1D array of bytes. Each byte has an address.

MIPS handle data in words (1 word = 4 bytes = 32 bits).

MIPS use **aligned addressing** so the address of a word is always a multiple of 4.

MIPS Instructions

3 types of instructions:

- R-type: `op rd, rs, rt` (31-26 op, 25-21 rs, 20-16 rt, 15-11 rd, 10-6 shamt, 5-0 funct)
- I-type: `op rt, rs, imm` (31-26 op, 25-21 rs, 20-16 rt, 15-0 imm)
- J-type: `op imm` (31-26 op, 25-0 imm)

MIPS program labels:

- `.data` - start of data segment (variable declarations)
- `.text` - start of text segment (instructions)
- `.byte n` - 1 byte or char
- `.word n` - 4 bytes or int
- `.asciiz str` - null-terminated string
- `.space n` - preallocate n bytes of space in memory

Arithmetic Instructions

Type	Instruction	Meaning	Remarks
R	`add rd, rs, rt`	`rd = rs + rt`	
I	`addi rt, rs, imm`	`rt = rs + imm`	
R	`sub rd, rs, rt`	`rd = rs - rt`	
R	`mult rs, rt`	`HI:LO = rs * rt`	`HI` (higher 32 bits), `LO` (lower 32 bits)
R	`div rs, rt`	`rs = LO / rt`, `rt = LO % rt`	`LO` (quotient), `HI` (remainder)
R	`mfhi rd`	`rd = HI`	
R	`mflo rd`	`rd = LO`	
R	`move rd, rs`	`rd = rs`	Psuedo-instruction, `add rd, rs, $zero`

Type	Instruction	Meaning	Remarks
R	`neg rd, rs`	`rd = -rs`	Psuedo-instruction, `sub rd, $zero, rs`
R	`sll rd, rt, shamt`	`rd = rt << shamt`	Shift left logical
R	`srl rd, rt, shamt`	`rd = rt >> shamt`	Shift right logical, fill with 0
R	`sra rd, rt, shamt`	`rd = rt >> shamt`	Shift right arithmetic, fill with sign bit

Note: Operands are by default 32-bit signed integers.

To make 1-bit logical operations:

- `not` can be implemented by `xori 1`
  - `1 xori 1 = 0x0000 = 0`
  - `0 xori 1 = 0x0001 = 1`
- `nor` can be implemented by `nor` + `andi 1`
  - `0 nor 0 = ~(0x0000 | 0x0000) = 0xFFFF = -1`, `0xFFFF andi 1 = 0x0001 = 1`
  - `0 nor 1 = ~(0x0000 | 0x0001) = 0xFFFE = -2`, `0xFFFE andi 1 = 0x0000 = 0`
  - `1 nor 1 = ~(0x0001 | 0x0001) = 0xFFFE = -2`, `0xFFFE andi 1 = 0x0000 = 0`

Data Transfer Instructions

A complete clock cycle under MIPS:

0. PC holds the address of the next instruction, which is going to be executed this cycle
1. When the clock ticks, the instruction is fetched from memory
2. Control unit decodes the instruction, and send flags to the ALU and register file
3. Register file provides operands to the ALU, ALU executes the instruction, result is sent back to the register file
4. Control unit updates the PC = PC + 4 (1 instruction = 1 word = 4 bytes)

Type	Instruction	Meaning	Remarks
I	`lw rt, imm(rs)`	`rt = MEM[imm + rs]`	Load word from memory to register
I	`sw rt, imm(rs)`	`MEM[imm + rs] = rt`	Store word from register to memory
I	`li rt, imm`	`rt = imm`	Psuedo-instruction, `addi rt, $zero, imm`
I	`la rt, label`	`rt = label`	Psuedo-instruction

Where `rs` is base memory address, `imm` is offset.

ALU can communicate with memory through the data bus. In `lw` and `sw`, the ALU calculates the address of the memory location `imm + rs`, and the data bus transfers the data between memory and register.

Branch Instructions

Conditional branch instructions:

Type	Instruction	Meaning	Remarks
R	`slt rd, rs, rt`	`rd = (rs < rt) ? 1 : 0`	Set on less than
I	`slti rt, rs, imm`	`rt = (rs < imm) ? 1 : 0`	Set on less than immediate
I	`beq rs, rt, label` (*)	`if (rs == rt) PC = label`	Branch on equal
I	`bne rs, rt, label`	`if (rs != rt) PC = label`	Branch on not equal
I	`bnez rs, label`	`if (rs != 0) PC = label`	Branch on not equal zero

(*): the `label` parameter can be a label or an immediate value

- label: labels in the source code like `loop:`, `exit:`, etc. They are replaced by relative addresses during assembly
- immediate value: `beq rs, rt, offset` means: `if (rs == rt) PC = PC + 4 + offset * 4`

Unconditional branch instructions:

Type	Instruction	Meaning	Remarks
J	`j label`	`PC = label`	Jump
J	`jal label`	`ra = PC + 4; PC = label`	Call subroutine, so later can return to the next instruction (PC + 4)
J	`jr rs`	`PC = rs`	Subroutine return

Note: Function parameters are passed through registers `$a0-$a3`, and function return value is stored in register `$v0-$v1`.

Note: In case of recursion, all these registers will be overwritten. So we need to save them in the stack before calling another function.

Example: Calculate Fibonacci number Reference

```
.text
main:
    li $a0, 10
    jal fib # call fib(10)

    # result is stored in $v0
    move $a0, $v0
    li $v0, 1
    syscall
    j exit
fib:
    # fib(n) = fib(n-1) + fib(n-2)
    # for each fib(n) call, we need to memorize 3 values:
    # 1. ra (return address)
    # 2. n
    # 3. result of fib(n-1)
    ble $a0, 1, fib_end # if n <= 1, return n

    # allocate 3 words in stack
    addi $sp, $sp, -12
    sw $ra, 0($sp) # save ra
    sw $a0, 4($sp) # save n

    # call fib(n-1)
    addi $a0, $a0, -1 # n-1
    jal fib
    sw $v0, 8($sp) # save fib(n-1)

    # call fib(n-2)
    lw $a0, 4($sp) # restore n
    addi $a0, $a0, -2 # n-2
    jal fib

    lw $t0, 8($sp) # load fib(n-1)
    add $v0, $v0, $t0 # fib(n) = fib(n-1) + fib(n-2)
    lw $ra, 0($sp) # restore ra
    addi $sp, $sp, 12
    jr $ra

fib_end:
    move $v0, $a0 # return n
    jr $ra

exit:
    li $v0, 10
    syscall
```

Writing a for loop:

```
for (int i = 0; i < 10; i++) {
    // do something
}
```

```
    li $t0, 0 # i = 0
    li $s0, 10 # loop count = 10

loop:
    # do something
    addi $t0, $t0, 1 # i++
    bne $t0, $s0, loop # if (i != 10) goto loop
    j exit
```

5. CPU

Structure:

- State elements: PC, Register File, Instruction Memory, Data Memory
- Control Logic: Reads opcode and generates control signals
- Multiplexers: Selects between different inputs or outputs based on control signals

Register file:

- Inputs: Read register 1, Read register 2, Write register, Write data, Write enable
- Outputs: Read data 1, Read data 2
- Register file is implemented by decoder (to select write register) + multiplexer (to select read data)

Control Logic

- RegDst:
  - 1: use `rd` (bits 15-11) as write register (for R-type)
  - 0: use `rt` (bits 20-16) as write register (for I-type)
- RegWrite:
  - 1: allow write to register file
- ALUSrc:
  - 1: use sign-extended immediate value as second ALU input (for I-type)
  - 0: use read data 2 as second ALU input (for R-type)
- PCSrc (Branch):
  - 1: use branch target address as next PC
  - 0: use PC + 4 as next PC
- MemRead:
  - 1: read data from memory
- MemWrite:
  - 1: write data to memory
- MemtoReg:
  - 1: use read data from memory as write data to register file (for `lw`)
  - 0: use ALU result as write data to register file
- ALUOp (2 bits)

Instruction	`add`	`addi`	`lw`	`sw`	`beq`	Remarks
RegDst	1	0	0	X	X	

Instruction	`add`	`addi`	`lw`	`sw`	`beq`	Remarks
RegWrite	1	1	1	0	0	
ALUSrc	0	1	1	1	0	`lw` and `sw` will calculate memory address
PCSrc	0	0	0	0	1	
MemRead	0	0	1	0	0	
MemWrite	0	0	0	1	0	
MemtoReg	0	0	1	X	X	

Note: `sw` and `beq` does not write to register file, so related fields are don't care.

ALU

Input of ALU Control:

- 2-bit ALUOp from control logic
- 6-bit funct at the LSB of R-type instruction

Output of ALU Control: 4-bit ALU control signal, which decides the arithmetic operation of ALU

Pipeline

Above model of a single-cycle CPU is too slow. The CPU has to wait for the slowest instruction to finish.

Pipeline divides instructions into 5 stages:

- Instruction Fetch (IF): fetch instruction from memory
- Instruction Decode (ID): decode instruction and read register file
- Execute (EX): execute ALU operation
- Memory (MEM): access memory
- Write Back (WB): write result to register file

So that each stage of a single instruction can be executed by different components sequentially, and different stages of different instructions can be executed in parallel.

Pipeline registers are used to store intermediate results between stages.

Structural Hazard

When two instructions need to use the same hardware resource at the same time. (e.g. IF and MEM both need to access memory)

Solution: Separate instruction memory and data memory.

Data Hazard

When one instruction depends on the result of another instruction that is still in the pipeline.

Solution: Forwarding (bypassing)

Use multiplexers to transfer the result of ALU to the next stage of the pipeline.

Forwarding cannot completely avoid pipeline stalls.

Control Hazard

When the next instruction to be executed is not the next instruction in the program.

Branch prediction: a hardware mechanism that tries to guess the next instruction to be executed.

Simple branch prediction: always predict not taken. If the prediction is right, the next instruction can be executed immediately. If the prediction is wrong, the pipeline stalls.

Dynamic branch prediction: to look up the address of the next instruction in a branch prediction table, check if it was taken last time, and predict the same result.

2-bit branch prediction: a prediction must fail twice in a row to change the prediction.

6. Memory

- Spatial locality: if a memory location is referenced, it is likely that nearby memory locations will be referenced soon
- Temporal locality: if a memory location is referenced, it is likely that the same memory location will be referenced again soon

Programs often use the same or related data during a short period of time, so it is important to store the data in a fast memory (cache) to improve performance.

Memory Hierarchy

Registers > L1 Cache > L2 Cache > L3 Cache > Main Memory > Secondary Storage

Direct Mapped Cache

Consider in a 32-bit system:

- a main memory of 2^M bytes (= 2^{M-2} blocks x 4 bytes per block)
- a cache of 2^C bytes (= 2^{C-2} lines x 4 bytes per line)

A cache refering to a main memory value at address m:

- Cache index = m mod 2^{C-2}
- Cache tag = m div 2^{C-2}
- Offset = m mod 4

A cache line consists of 3 fields:

- 1 valid bit
- 1 tag field, length = 32 bits - cache index length ($C-2$ bits) - offset length (2 bits) = $32-C$ bits
- n data fields, length = 32 bits, n = words per line = bytes per line / 4 (1 in this case)

Example: a main memory of 16 KB (= 4096 blocks x 4 bytes per block), a cache of 256 bytes (= 64 lines x 4 bytes per line)

- Cache index length = 6 bits
- Offset length = 2 bits
- Tag length = 32 bits - 6 bits - 2 bits = 24 bits

- Address `0x0000`: cache index = 0 mod 64 = 0, cache tag = 0 div 64 = `0x000000`, offset = 0 mod 4 = 0: access cache line 0, offset 0
- Address `0x0004`: cache index = 1 mod 64 = 1, cache tag = 1 div 64 = `0x000000`, offset = 1 mod 4 = 1: access cache line 1, offset 1
- Address `0x07FF = 0b11111111111`: cache index = `0b111111111` mod 64 = 63, cache tag = `0b111111111` div 64 = `0b111` = `0x000007`, offset = `0b11` mod 4 = 3: access cache line 63, offset 3
- Address `0x0800`: cache index = `0b10000000000` mod 64 = 0, cache tag = `0b10000000000` div 64 = `0b1000` = `0x000008`, offset = `0b00` mod 4 = 0: access cache line 0, offset 0

Fully Associative Cache

A memory block can be stored in any cache line.

Tag length = 32 bits - offset length (2 bits) = 30 bits

To check if an access is a cache hit, need to check all cache lines. (use a multiplexer)

When a cache miss occurs, the memory value will be loaded into the cache line. **Replacement policy** is required to decide which cache line to be replaced.

- **FIFO**: first in first out (when miss occurs, load the new value and set age to 0, increase age of all other cache lines by 1)
- **LRU**: least recently used (when miss occurs, load the new value and set age to 0, increase age of all other cache lines by 1; when hit occurs, set accessed cache line age to 0, increase age of all other cache lines by 1)

Set Associative Cache

The memory blocks with same cache index does not go to a single cache line, but a set of cache lines.

A set associative is called n-way associative if each set contains n cache lines.

Suppose a 32-bit system:

- a main memory of 4 GB (= 2^{32} bytes = 2^{30} blocks x 4 bytes per block)
- a cache of 16 KB (= 2^{14} bytes = 2^{10} sets x 4 lines per set x 4 bytes per line)

A cache refering to a main memory value at address m:

- Cache index = $\frac{m}{4}$ mod 2^{10} (10 bits)
- Cache tag = $\frac{m}{4}$ div 2^{10} (20 bits)
- Offset = m mod 4 (2 bits)

To locate a memory value in cache, need to check all cache lines in the set. Replacement policy is used within each set.

For example,

- address `0x000C`: cache index = 3 mod 1024 = 3, cache tag = 3 div 1024 = 0, offset = 12 mod 4 = 0: access cache set 3
- address `0x400C`: cache index = 4099 mod 1024 = 3, cache tag = 4099 div 1024 = 4, offset = 16396 mod 4 = 0: access cache set 3

A 1024-set 4-way associative cache consists of 4 cache storage, each storage contains 1024 cache lines. Set 3 consists of cache line 3 from each storage.

To read a memory value, the 4 results from 4 cache lines are all read and sent to multiplexer. The multiplexer selects the correct result based on the cache tag.

In a 32-bit system of 4GB (= 2^{30} bytes) main memory and 16KB (= 2^{12} bytes) cache:

- Direct mapped cache: bits 31-14 (18 bits) are cache tag, bits 13-2 (12 bits) are cache index, bits 1-0 (2 bits) are offset
- Fully associative cache: bits 31-2 (30 bits) are cache tag, bits 1-0 (2 bits) are offset
- 4-set associative cahce (1024 sets x 4 lines per set): bits 31-12 (20 bits) are cache tag, bits 11-2 (10 bits) are cache index, bits 1-0 (2 bits) are offset

SRAM vs DRAM

SRAM (Static Random Access Memory):

- Use flip-flops to store data
- In cache, register file, etc.

DRAM (Dynamic Random Access Memory):

- Use capacitors to store data
- In main memory
- Volatile (data is lost when power is off)
- More power consumption (need to refresh data periodically)

Storage Devices

HDD (Hard Disk Drive):

- Magnetic storage
- **Non-volatile** (data is not lost when power is off)

SSD (Solid State Drive):

- Flash memory
- Non-volatile

ROM (Read Only Memory):

- Can be written only once
- Non-volatile
- Use to store firmware (BIOS, boot loader, etc.)

NAND Flash Memory

- NAND flash memory is organized into blocks
- A block consists of multiple pages
- A page consists of cells spaced in a grid, that can be located by row and column
- Block is the unit of erase operation (erased cells are set to 1)
- Page is the unit of read and write operation (write 0 to a cell will change the cell to 0, but write 1 to a cell will not change the cell to 1)

Flash cell:

- Floating gate transistor
- Stores charge in the floating gate

- Low charge: 1 (erased), high charge: 0 (programmed)
- **Read voltage**: the voltage applied to the cell to read the charge
    - If the cell is high charge, the read voltage will be drained
    - If the cell is low charge, the read voltage will be passed to the bit line
- Can only charge a single cell (1 -> 0), but not discharge a single cell (0 -> 1)
- To change a cell from 0 to 1, first erase the block

How to read a specific page (a row of cells) in a block:

- Add read voltage to the selected page
- Add pass voltage (e.g. 5V, larger than read voltage) to the unselected pages
- Unselected pages will just pass the pass voltage to the bit line (going down a column)
- When the pass voltage reaches the selected page
    - If the cell is high charge (0), it will drain the pass voltage, so the result voltage on the bit line will be low (0)
    - If the cell is low charge (1), nothing will happen, so the result voltage on the bit line will be high (1)

7. I/O

I/O controller connects to:

- the system bus (address, data, control)
- peripheral devices (an interface to keyboard, mouse, etc.)

The external interface (port) connects I/O controller to peripheral devices.

It

- receives control signals from I/O controller
- sends control signals to I/O controller
- Use a buffer to forward data between I/O controller and devices

I/O addressing:

- **Memory-mapped I/O**: I/O transfer is treated as read/write a section of memory
- **Isolated I/O**: Each device has a unique address space, and special instructions are used to access the device

Read CS2115_extra.md for more details.

8. Parallelism

Processing Models:

- Single instruction, single data (SISD): as in a single-core CPU
- Single instruction, multiple data (SIMD): as in a GPU

Parallelism: use multiple processing units to execute multiple instructions at the same time

The speedup is limited by **Amdahl's Law**: assume p portion of the program can be parallelized, then the maximum speedup is $\frac{1}{1-p}$.

GPU (Graphics Processing Unit): a SIMD processor that is optimized for graphics processing.

Due to the nature of graphics processing, many of the work is similar but just applied to different data. So GPU is designed to execute the same instruction on multiple data at the same time.

GPU is SIMD. It fetches a single instruction from the instruction memory, and executes the instruction on multiple data at the same time.

GPU is organized in hierarchical structure:

- **SP (Streaming Processor)**: smallest processing unit, executes a single instruction (thread) on multiple data at the same time
- **SM (Streaming Multiprocessor)**: a group of SPs, share a local memory (cache, SRAM)
- **GPU**: a group of SMs, all SMs share a global memory (GDDR RAM, DRAM)

CUDA is a parallel GPU programming API created by NVIDIA. The GPU does not necessarily use CUDA; other models like OpenGL and DirectX are also used to render graphics.

In CUDA programming:

- **Kernel**: a function that runs on GPU
- **Thread**: a single execution of a kernel, runs on a single SP (different threads in a kernel has the same instruction, but work on different data)
- **Warp**: a group of 32 threads, all threads in a warp execute the same instruction at the same time
- **Block**: a group of threads, all threads in a block are executed on the same SM (threads in a block can communicate with each other)
- **Grid**: a group of blocks, the largest unit of CUDA runtime, each kernel corresponds to a grid (so a function can be run on multiple SMs)

Each block and warp has a unique ID, decided by its relative position in the grid/block, respectively.

CUDA programming keywords:

- `__host__` is only compiled by Host (CPU)
- `__global__` is only compiled by Device (GPU)
- `__device__` is only compiled by Device (GPU), and cannot be called by Host (CPU) (i.e. device-private)
- `__shared__` is compiled by Host (CPU) and Device (GPU)
- `kernel <<<grid, block>>>` (executed by CPU) launches a kernel on GPU

Example 1: Find the maximum value of an array of N integers ($N \leq 2^{20}$).

We allow the program to be executed on 16 SMs (16 blocks in the grid), each SM has 32 SPs (32 threads in a block).

```
__device__ int max(int a, int b) {
    return a > b ? a : b;
}

__global__ int find_max(int *arr, int N, int *result) {
    int kid = blockIdx.x * blockDim.x + threadIdx.x; // kernel ID
    // each kernel will check elements at index idx where idx % 512 == i
    for (int i = kid; i < N; i += gridDim.x * blockDim.x) { // gridDim.x = 16,
blockDim.x = 32
        result[kid] = max(result[kid], arr[i]);
    }
}

// in main()
find_max <<<16, 32>>> (arr, N, result);
```

Example 2: 2D matrix addition.

Each SM has 16 x 16 = 256 SPs (256 threads in a block), and we allow any number of SMs.

```
__global__ void mat_add(float *A, float *B, float *C, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N) {
        C[i][j] = A[i][j] + B[i][j];
    }
}

// in main()
dim3 dimBlock(16, 16);
dim3 dimGrid((N + dimBlock.x - 1) / dimBlock.x, (N + dimBlock.y - 1) /
dimBlock.y);
mat_add <<<dimGrid, dimBlock>>> (A, B, C, N);
```

Tradeoff of GPU:

- Very high throughput
- Very weak single-thread performance (and no pipelining, no branch prediction, etc.)
- Programs unrelated to graphics processing can also benefit from GPU, but need to be highly parallelized

Connection Networks

- **Bus**: a single shared connection between all nodes
    - Bus can only be used by one node at a time
- **Ring**: a single connection that connects all nodes in a ring
    - Bidirectional ring: data can be sent in both directions, halving the latency
    - Hierarchical ring: multiple rings are connected together, reducing the number of hops
- **Crossbar**: a fully connected network, direct connection between any two nodes

Shared Memory

- **Uniform Memory Access (UMA)**: memory is after the connection network, all nodes can access memory with the same latency
- **Non-Uniform Memory Access (NUMA)**: memory is before the connection network, distributed between nodes, nodes can access local memory with lower latency

Comparison of UMA and NUMA:

- UMA is easier to implement
- UMA use a single memory controller, NUMA requires a memory controller for each node
- NUMA is usually faster than UMA
- NUMA has higher bandwidth than UMA

Cache Coherence

Issue: when multiple processors cache the same memory block, how to ensure that the data in the caches are consistent?

Snoopy cache: all caches are connected to a shared bus, monitor other nodes' write operations on cached memory blocks, and invalidate the corresponding cache lines.

Main issue: broadcast consumes a lot of bandwidth.

Directory-based cache: a logically central directory records which records "which nodes has cached which memory blocks".

Define: **Owner**: the node that most recently modified the memory block (and thus holds the most recent copy). **Sharer**: the node that has a copy of the memory block.

- Upon receiving a read request, the directory will forward the request to the owner, and the owner will return the data to the requester
- Upon receiving a write request, the directory will inform all sharers to invalidate the cache line

Comparison of snoopy cache and directory-based cache:

- Snoopy cache use full broadcast, directory-based cache use point-to-point communication (or limited broadcast)
- Snoopy cache is simpler to implement
- Snoopy cache has lower latency
- Directory-based cache has higher bandwidth
- Directory-based cache is more scalable

Message Passing

In shared memory (UMA and NUMA), all nodes share the same memory address space.

In Message Passing, each node has its own memory address space, and each memory is exclusive to the local node. Different nodes can only communicate through message passing.

Comparison of shared memory and message passing:

- Shared memory uses a common address space, message passing uses different address spaces
- Shared memory is more efficient
- Shared memory is easier to program
- Shared memory requires Cache Coherence, message passing does not allow multiple nodes to access the same memory block, so no consistency issue

Appendix A. Pipelining

Based on CUHK CSCI 2510 Lec 11

In the textbook, the pipeline is divided into 5 stages:

- Instruction Fetch (IF):
  - a MUX between PC+4 and PC+4+4*offset, to update PC
  - a register to store the PC
  - the instruction memory

  - an adder to calculate PC+4
- Instruction Decode (ID):
  - breaks the instruction and sends to different components
  - the control logic which sends control signals to EX, MEM, WB
  - the sign extender to extend the immediate value
- Execute (EX):
  - a MUX between read data 2 and sign extended immediate value, input = **ALUSrc** (from control logic), to select second ALU input
  - the ALUOp which take funct and **ALUOp** (from control logic) as input, and output ALU control signal
  - a MUX between bits 20-16 (rt) and bits 15-11 (rd), input = **RegDst** (from control logic), to select write register
  - the ALU
  - calculate PC+4+4*offset
- Memory (MEM):
  - a **MemRead** signal (from control logic) to read data from memory
  - a **MemWrite** signal (from control logic) to write data to memory
  - send back PC+4+4*offset, and the result of **branch** condition (checked by ALU) to IF
- Write Back (WB):
  - a MUX between ALU result and read data from memory, input = **MemtoReg** (from control logic), to select write data

Structural Hazard

When two instructions need to use the same hardware resource at the same time. (e.g. IF and MEM both need to access memory)

Solution: Separate instruction memory and data memory.

In practice, it is "instruction cache" and "data cache".

Data Hazard

When one instruction depends on the result of another instruction that is still in the pipeline.

Solution: Use **forwarding** (bypassing) to transfer the result of ALU to the next stage of the pipeline.

1. `rs`, `rt` of the next instruction are forwarded from ID to EX
2. ALU result of current instruction is forwarded from MEM to EX
3. Memory read data is forwarded from WB to EX
4. The forwarding unit will send proper fields as ALU operands in EX stage

Forwarding cannot completely avoid pipeline stalls.

e.g.

```
lw $s0, 20($t1)
sub $t2, $s0, $t3
```

By forwarding unit, the ALU can directly use the result of `lw` to execute `sub`. However, the value is not known until the end of MEM stage.

Therefore, EX stage of `sub` needs to be stalled until MEM stage of `lw` is finished.

Moderm processors use **out-of-order execution** to give memory access instructions more time to execute, but cannot completely avoid pipeline stalls.

Control Hazard

When the next instruction to be executed is not the next instruction in the program.

Branch penalty: the number of cycles wasted due to incorrect branch prediction (such that some instructions are partly executed but not committed)

Solution: Instruction prefetching, branch prediction

Instruction Queue: a buffer that stores instructions that are fetched, decoded, but not executed yet.

Branch prediction: a hardware mechanism that tries to guess the next instruction to be executed.

Simple branch prediction: always predict not taken. If the prediction is right, the next instruction can be executed immediately. If the prediction is wrong, the pipeline stalls.

Dynamic branch prediction: to look up the address of the next instruction in a branch prediction table, check if it was taken last time, and predict the same result.

1-bit branch prediction:
- buffer 1 bit for each branch instruction (e.g. `beq`)
- If branch is used in a loop, the prediction will always be incorrect for 2 times
  - First time: Last time some loop terminates, so the prediction is not taken
  - Last time: Inevitable

2-bit branch prediction:
- buffer 2 bits for each branch instruction
- Has 4 states (Strongly Taken = 11, Weakly Taken = 10, Weakly Not Taken = 01, Strongly Not Taken = 00)
- If branch is used in a loop, the prediction will always be incorrect for 1 time
  - First time: Last time some loop terminates, the prediction change from Strongly Taken to Weakly Taken
  - Last time: Inevitable

Example of forwarding:

```
lw $5, 10($6)
add $7, $8, $9
sub $10, $5, $7
lw $11, 10($10)
sw $11, 10($12)
```

	1	2	3	4	5	6	7	8	9	10
lw	IF	ID	EX	**MEM**	WB					
add		IF	ID	**EX**	MEM	WB				
sub			IF	ID	**EX**	MEM	WB			

	1	2	3	4	5	6	7	8	9	10
lw				IF	ID	**EX**	**MEM**	WB		
sw					IF	X	ID	**EX**	MEM	WB

For `lw`: the value of `$5` can be forwarded after MEM stage (t=4).

For `add`: the value of `$7` can be forwarded after EX stage (t=4).

For `sub`: the EX stage rely on `$5` and `$7`, so it can only be executed 1 cycle after `lw` MEM and `add` EX (t=5), with the instruction began at t=3. The value of `$10` can be forwarded after EX stage (t=5).

For `lw`: the EX stage rely on `$10`, so it can only be executed 1 cycle after `sub` EX (t=6), with the instruction began at t=4. The value of `$11` can be forwarded after MEM stage (t=7).

For `sw`: the EX stage rely on `$11`, so it can only be executed 1 cycle after `lw` MEM (t=8), with the instruction began at t=6. However, the instruction is actually scheduled at t=6, so a stall of 1 cycle happens after IF, before ID (t=7).