Pipelining and Hazards

Structural Hazards: Structural Hazards occur when more than one instruction needs to use the same datapath resource at the same time. In the 5-stage pipeline, there aren't structural hazards, unless there are actual changes to the pipeline.

There are two main causes of structural hazards:

→ RegFile: The RegFile is accessed at both Instruction Decode (when it is read for instruction decode and reg values) and during Writeback (when it is written to in the rd register). If the RegFile had one port, then it wouldn't work since one instruction is being decoded and another writing back) We fix this with seperate read/write ports. However, read/write register must be different

→ Main Memory: Main memory is accessed for both instructions and data. If memory could only support one read/write at a time, then instruction A going through IF and attempting to fetch from memory cannot happen at the same time as Instruction B attempting to read (or write) to data portions of memory. Having a seperate instruction memory (IMEM) and data memory (DMEM) solves this hazard.

Structural Hazard can always be resolved by adding more hardware.

Data Hazards

Data Hazards are always caused by data dependancies between instructions.

Three types of data hazards:

→ EX-ID: this hazard exists because the output from the execute stage is not written back to the RegFile until the WriteBack stage, yet can be requested by the subsequent instruction in the decode stage.

→ MEM-ID: this hazard exists because the output from the memory access is not written back to the RegFile until the writeback stage, but can be requested from the decode stage, just as in EX-ID

→ WB-ID: To account for reads and writes to the same register processors usually write to the register during the first half clock cycle, and read from it during the second half.

We can resolve most Data Hazards by forwarding, which is when the result of the EX or MEM stage is sent to the EX stage for a following instruction to use. We can also use stalls by inserting a nop (no operation) between the instructions.

Control Hazards

Control Hazards are caused by jump and branch instructions, because for all jumps and branches, the next PC is not PC+4, but the result of the ALU available after EX stage. We could stall the pipeline for control hazards, but this decreases performance.

Data hazard: Reg Access

```
add t0 t1 t2  IF ID EX MEM WB
  instruc        IF ID EX MEM WB
  instruc           IF ID EX MEM WB
sw t0 4(t3)            IF ID EX MEM WB
```

Same register is written and read in one clock cycle ⇒ WB must write value before ID reads new value

Solution: RegFile hardware should enable write-then-read in same cycle also known as double pumping

Data Hazard: ALU Result

```
add s0 t0 t1  IF ID EX MEM WB
sub→nop          IF  #  #  #  #
sub→nop             IF  #  #  #  #
sub t2 s0 t0           IF ID EX MEM WB
```

Sub instruction depends on prev WB's RegFile write. Executing sub immediately reads old value of s0. This solution stalls the pipeline for two clock cycles.

Data Hazard: Forwarding

```
add s0 t0 t1  IF ID EX MEM WB
sub t2 s0 t0     IF ID EX MEM WB
or  t6 s0 t0        IF ID EX MEM WB
```

Forwarding uses the result when it is computed Doesn't wait for value to be stored in RegFile And grabs operand from pipeline immediately

Loads however result in an unavoidable pipeline stall because the result isn't ready till the finish of DMEM, not in time to be forwarded to the following instruc's ALU/EX stage. Stall once, then use forwarding or use code scheduling.

Control Hazard: Conditional Branches / Jump

If the branch is taken, kill all instructions after the branch (if taken) by converting incorrect instructions to nops. This then means that every taken branch takes 3 clock cycles. We can use branch predictor and flush if they were incorrect.

Examples:

data hazard: line 1-2, 1-3

```
1. addi t0  a0  -1
2. andi s2  t0  a0
3. sltiu a0 t0  5
```

Forward 1's EX to 2's EX and 1's Mem to 3's EX.

Data hazard: line 2-3, 3-4

```
1. addi s0 s0 1
2. addi t0 t0 4
3. lw   t1 0(t0)
4. add  t2 t1 x0
```

forwards 2's EX to 3's EX and insert nop between 3 and 4 and forward 3's MEM to 4's EX

Without double pumping, 3 instructions after can be affected by data hazard and 2 with it since the ID of instruction 4 could be lined up with the WB of instruc 1.

Caches

Cache Hit: The data you were looking for is in the cache. You retrieve the data from the cache and bring it to the processor

Hit Rate: fraction of access that hit the cache. Hit time: time (latency) to access cache.

Cache miss: The data you were looking for is not in the cache. Go to a lower layer in the memory hierarchy to find the data and put the data in the cache. Then bring the data to the processor.

Miss rate: 1 - Hit rate, Miss penalty: time (latency) to replace block from lower level in memory hierarchy to cache.

Average Memory Access Time: Hit time * Hit rate + Miss rate * (Hit time + Miss penalty)

Fully Associative Cache (Put a block - # of bytes in a row - anywhere)

Placement Policy: data is stored anywhere in the cache.

For any given memory address, split it into tag and offset.

When starting a program, the cache may not have valid info for program necessitating the need for a valid bit.

Terminology: Cache block/line: A single entry in the cache, Block size/line size: # of bytes per cache block, Cap[illegible]tal # data bytes that can be stored in a cache, Tag: Identifies do[illegible]a given cache block.

Math: # offset = $\log_2$(block size); # [illegible]s bits - offset; Capacity = # lines * line size

Block replacement policy

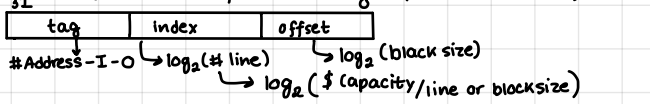
When FA cache reaches capacity, we can miss, therefore we need block replacement. Nice way to do so is LRU which replaces the entry that has not been used for the longest time but this is hard to do/complicated. Theres also MRU, FIFO, LIFO, and Random.

Approximations to LRU and MRU

Write policy

Store instruction writes to memory which changes value so we need to insure that cache and memory is consistent. We can use write-through which writes to the cache and memory at the same time or write-back where we write data in cache, set dirty bit to 1, and write updated data in cache to memory when corresponding block is evicted

Direct Mapped Cache (Each memory address has one possible block in $)



This is much simplier then FA cache since we just need to check one tag. Additionally, there isn't a replacement policy; you just replace whatever is at the index when $ miss occurs.

Types of Miss

Compulsory Miss: Caused by first access to a block that has never been in the cache

Capacity Miss: Caused when the cache cannot contain all the blocks needed during progam execution.

Conflict Miss: Multiple blocks compete for the same location in the cache, even when the cache has not reached full capacity

Recursive AMAT: L1 HT + L1MR * (L2 HT + L2 MR * MMA)

Parallelism:

Amdahl's Law: 1/[(1-F)+F/S] = speed-up where F = fraction of program being sped up and S = speed up factor.

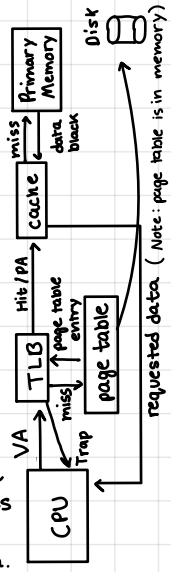
```
Thread 1
lw t0 0(s0)
beq t0 x0 equal
j done
equal: lw t0 0(s0)
addi t0 t0 1
sw t0 0(s0)
jal printf

Thread 2
lw t0 0(s0)
beq t0 x0 equal
j done
equal: lw t0 0(s0)
addi t0 t0 1
sw t0 0(s0)
jal printf
```

```
#pragma omp parallel
{
  if (x==0){
    x=x+1;
    printf(...);
  }
}
```

What should we do if main memory is smaller than physical address space? What if two programs access the same memory address?
These two cases can be resolved with Virtual memory.
Virtual memory gives each process the illusion of a full memory address space. These processes use virtual addresses. While each process thinks it has the entire address space to work with, it does not. All processes share the same physical memory but there is a translator which translates these virtual addresses to physical addresses.

Virtual address space: Set of addresses that the user knows about (Translation takes place)
Physical address space: Set of addresses that map to physical locations in mem

OS responsibilities:
- The OS needs to map VAs to PAs
- Since we want to give the illusion of a full address space, we'll need to store some content of disk rather than DRAM and we need the OS to isolate this memory for protection.



Paged Memory: Break our physical memory into multiple regions called **pages**. Since we divide memory into (usually 4 KiB byte) chunks, this introduces the notion of **page offset** to address each byte in our page.
Hence we can use this system of **pages and page offset** for memory address translation.

VA: Virtual page number (20 bits)	Offset (12 bits)

TRANSLATION!!!

PA: Physical page number (36 bits)	Offset (12 bits)

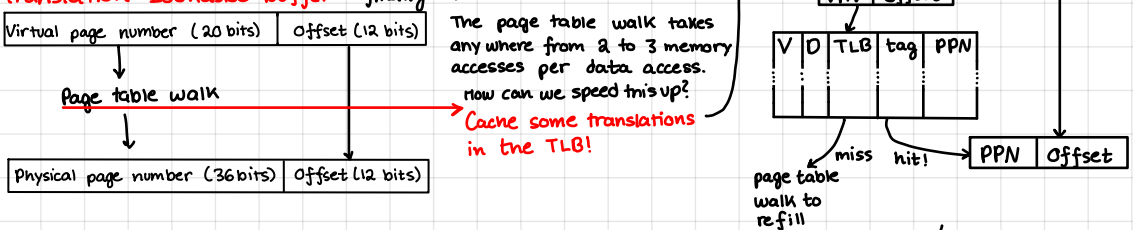
How do we **translate**?
The program wants to load something at a VA. So it asks the computer to translate the VA to a PA in memory by extracting the top 20 bits from the VA (which is our VPN) and looks up the corresponding **PPN** in a **page table**. We can construct the **PA** from the **PPN and page offset**. If the physical page is not in memory, a **page fault occurs** and the OS loads the page from the disk. If a **page fault** occurs, we load the page from disk into memory and then update the page table to reflect the new physical page it is at. Then proceed as normal.

Page table details:
- Each process gets its own page table and these page tables are stored in memory!! (Pages tables are way too large to store in a cache. → As a result, we need to read page table and physical page. (2 memory accesses in total))
- If the virtual address space is 32 bits and the page size is 4 KiB, then there are $2^{32}/2^{12} = 2^{20}$ VPNs.

Math :(: (Assuming 32-bit machine with 8 GiB of RAM and 16 KiB pages)
Page offset = $\log_2$(page size); VPNs = #VA bits − page offset; PPNs = $\log_2$(physical memory size) = #PA bits − offset bits
$\log_2(16\text{ KiB}) = 14$ = 32 − 14 = 18 = $\log_2(8\text{ GiB}) = 19$

Page tables store **status bits** to show whether page is in memory or on disk only. If the status bit indicates valid ⇒ page is in memory else it is on disk which will trigger a page fault OS tries to allocate page into memory If memory is full, a page is evicted to the disk, stores page at evicted location, and updates page table. Page tables also have a **dirty bit** which indicates if the page on RAM is more up-to-date than disk and a **Write Protection bit** which triggers an exception if program attempts to write to this page.
There are 100s of processes running on a computer at any given moment. If there are multiple cores then the OS multiplexes over all **available cores** but if there is a single core then the OS performs a **context switch**. A context switch is when the OS switches out the state of the processor between processes (programs) through the use of the **trap handler**.
The trap handler is code that services exceptions and interrupts. It completes all instruction before the fault, flush everything after faulting instruction, and transfers execution to trap handler in **supervisor mode** (as opposed to **user mode** which only looks at a subset of instructions and memory.)
The trap handler will save the current state of the program, determines what causes the exception and handles exception. From here it either continue execution of the program or terminates it.
The OS sets a timer and when it expires, it performs a **hardware interupt**, and saves the PC along with the **page table register**.

Translation Lookaside Buffer finally :)



The TLB Reach is the size of largest VA space that can be simultaneously mapped by the TLB
The TLB is usually 38-128 entries big and is fully associative with a Random/FIFO replace policy
The TLB is indexed by the VPN where the VPN is broken up into a **TLB tag** and **TLB index**.

Last Minute Things:

Number Representation

Two's Complement: the standard for number representation. To represent a negative number, write the positive number in binary. Then flip the bits and add 1 to the least significant bit. The resulting binary string is the number in two's complement.

Bias notation: Treat the number as an unsigned, then add bias.
Example: -17 with 8 bits and a bias of -31
Formula: $x + b = n$ (x is the binary rep, b is bias, n is number we want to represent) $x - 31 = -17 \Rightarrow x = 14$ so represent 14 in 8 bit unsigned!

Hexadecimal: More compact representation of binary. Every 4 bits of binary corresponding 1 hex digit

N bits represent 2^N things
N bits unsigned : $[0, 2^N-1]$
N bits sign-magnitude : $[-(2^{N-1}-1), 2^{N-1}-1]$
N bits Two-complement: $[-2^{N-1}, 2^{N-1}-1]$
smallest: 1 sign bit followed by all zero; largest: 0 sign bit, followed by all 1

Converting into arbitrary base:
One's complement: Invert the bits of the positive representation in order to get the negative representation

Floating Point

Normalized Floats: Value = $(-1)^{sign} * 2^{Exp+Bias} * 1.significand_2$

Denormalized Floats: Value = $(-1)^{sign} * 2^{Exp+Bias+1} * 0.significand_2$

Exponent	Significand	Meaning
0	Anything	Denorm
1-254	Anything	Normal
255	0	Infinity
255	Nonzero	NaN

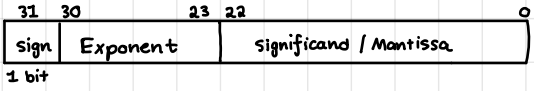
The bias for a single precision floating point number can be calculated using $-(2^{exp.bits-1}-1)$

Common powers of 2:

$2^1 = 2$	$2^{-1} = 0.5$
$2^2 = 4$	$2^{-2} = 0.25$
$2^3 = 8$	$2^{-3} = 0.125$
$2^4 = 16$	$2^{-4} = 0.0625$
$2^5 = 32$	$2^{-5} = 0.03125$
$2^6 = 64$	
$2^7 = 128$	
$2^8 = 256$	
$2^9 = 512$	
$2^{10} = 1024$	

A = 10
B = 11
C = 12
D = 13
E = 14
F = 15

Single Precision Representation



C Programming

pointers are always 4 bytes long
When considering sizeof(struct) make sure to consider memory alignment
sizeof(uint16_t) = 2 bytes; sizeof(uint32_t) = 4 bytes; sizeof(uint64_t) = 8 bytes

& (bitwise and between var1 and var2 and returns result of &)
| (bitwise or between var1 and var2 and returns result of |)
a << b (logical left shift: shifts a, b digits to the left by b digits)
a >> b (logical right shift: shifts a, b digits to the right by b digits)
~a (negates all digits of a); a^b (computes XOR between a and b)

The Stack: local variables inside functions, where data is garbage immediately after the function in which it was defined returns. Each function call creates a stack frame with its own arguments and local variables. The stack dynamically changes, growing downwards as multiple functions are called within each other (LIFO structure), and collapsing upwards as functions finish execution and return

The Heap: memory allocated by the programmer with malloc, calloc, or realloc. Used for data that needs to persist through function calls. Memory is only freed when programmer explicitly frees it.

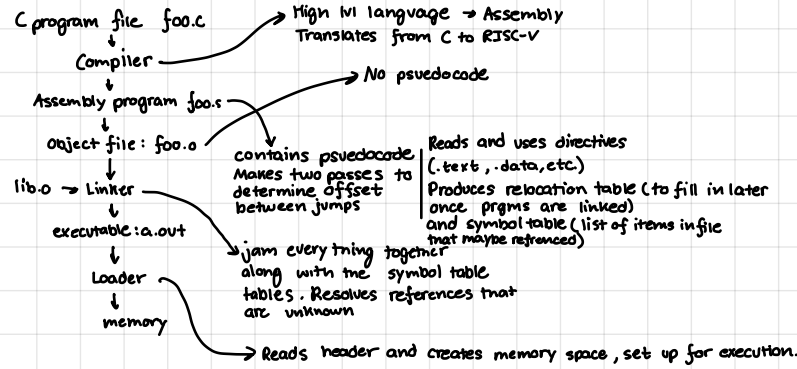
Static data: global variables declared outside of functions; does not grow or shrink during function execution; Code (or text): loaded at the start of the program and does not change after. contains executable instructions and any pre-processor macros.

malloc(size_t size); allocates a block of size bytes and returns start of the block. The time it takes to search for a block is generally not dependant on size
calloc(size_t count, size_t size); allocates a block of count*size bytes and sets every value in the block to zero, then returns the start of the block.
realloc(void *ptr, size_t size); "resizes" a previously-allocated block of memory to size bytes, and then returns the start of the resized block.
free(void *ptr); deallocates a block of memory which starts at ptr that was previously allocated by the previous three functions.

Generics and Function pointers

void * is a type that can point to anything
A function pointer is a variable storing the starting address of a function
Usually declared as retval (*fp)(params) and can be used by (*fp)(params).
If we have a function fp, we can access its address using the & operator (e.g. &fp)
C quirk: Outside of declaration, functions implicitly convert to pointers.
Dereferencing void * pointer gives us an error. To dereference a pointer, we must know the number of bytes to access from memory at compile time. Generics use generic pointer hence we cannot use the dereference operator (*).
To create temporary "generic" storage, declare local char array (i.e. buffer). This is because size of char == 1 byte
Additionally pointer arithmetics in generics must be bytewise arithmetic. Cast void* to char* so the pointer arithmetic is in fact byte-wise

Compiler, Assembler, Linker, and Loader



What are you doing step-size?

We can't accurately represent all number.

Ex: if we have $1.000_{10} * 10^4$ then 11,111 cannot be represented because $1.111_{10} * 10^4 = 11110$.

At some point, adding 1 will have no effect.

Conversion:

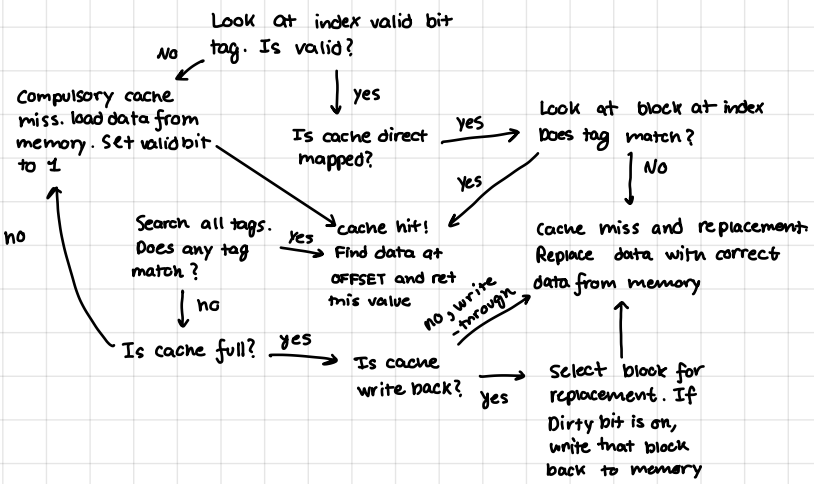
$+1.75_{10} * 2^{-129} \rightarrow +1.11_2 * 2^{-129} \rightarrow +1.11_2 * 2^{-126-3}$

$\rightarrow +0.00111_2 * 2^{-126}$

$\Rightarrow FP = 0x001C0000$

0b 1 0000 0000 010 0 ···
↳ $-0.01_2 * 2^{-126}$
↳ $-1.00_2 * 2^{-126-2} = -2^{-128}$

Cache Flowchart



RISC-V Programming

When dealing with strings, make sure to use lb instead of lw since characters are only a byte wide. We can use srai to divide negative and positive numbers whereas srli only works with positive numbers
If we want to compute modulus, then using andi rd, rs1, imm is good option

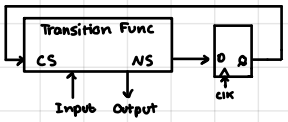
Instruction Formats

R-type instructions: Designed for instructions with 3 regs
add s2 s3 s4 to hex funct7:
Type: R; opcode: 0b011 0011; funct3: 0b000; 0b000 0000
Step 2: write out format
0b??????? ????? ????? ??? ????? ???????
Step 3: Registers
s2 → x18 → 0b10010
s3 → x19 → 0b10011
s4 → x20 → 0b10100
0b0000000 10100 10011 000 10010 0110011
Step 4: Convert to hex: 0x01498933
Translate 0x01B3 42BC to RV32 instruction
Step 1: Convert to binary and determine opcode and instruc type from reference card
0b 0000 0001 1011 0011 0100 0010 1011 0011
Opcode = last 7 bits = 0b011 0011 ⇒ R-type
Determine funct3/funct7 { funct3: 0b100, funct7: 0b000 0000 } ⇒ XOR operation
Step 3: Determine registers
rd: 0b00101 → x5 → t0
rs1: 0b00110 → x6 → t1
rs2: 0b11011 → x27 → s11
} xor t0 t1 s11

I-type: arithmetic operations with immediates, loads, jalr
I-type immediates are 12 bits. Most instructions use signed immediates therefore our range is [-2048, 2047]
I*-type: For shift instructions we have a max shift of 31, any larger shift will shift all the data out the #
S-type: Designed for instructions with 2 source registers and an immediate. We put rs1 and rs2 in same spot as R-type therefore we need to split immediate and then piece together
Consider li. How would we do li t0 0x12345678?
Split it up. lui t0 0x12345 and addi t0 t0 0x678
What about li t0 0xABCDEFFFF if we try to do lui t0 0xABCDE and addi t0 t0 0xFFF, 0xFFF ≠ 4095=1
Instead we need to do lui t0 0xABCDEF and addi t0 t0 0xFF
Dealing with offsets:
Each instructions takes 4 bytes of memory ⇒ offsets are multiples of 4.

Finite State Machines / Operating System

FSM circuit format:



Use a combinational logic block to recieve input and current state, and send output and next state. Save state in a register. FSM's are the same thing as Markov Chains.

The datapath we've learned is a CPU core. A single core can run one thread at any given time. Computer have multiple cores, hence they can run multiple threads. The OS coordinates processes and is run at startup.
User mode: limitations on what can be accessed but is safer to use. Kernel mode: has more freedom but can crash entire system.

Synchronous Digital Systems

$t_{hold} \leq t_{clk\text{-}to\text{-}Q} + t_{minimum}$

$t_{clk\text{-}period} \geq t_{clk\text{-}to\text{-}Q} + t_{maximum} + t_{setup}$

Both of the above are calculated from register to register.

Clock Period (T): in second or ns

Frequency (f): $f = 1/T$ (in Hz or GHz) (Hz = ticks per second)

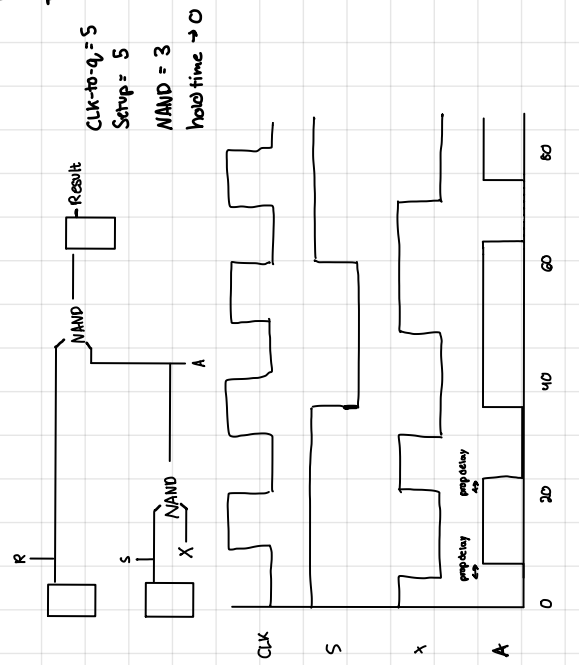
flip flop propogation delay = clk-to-q

We must account for propagation delays within our circuit.

Setup time: Amount of time that input needs to be stable before positive edge of the clk.

Hold time: Amount of time that input needs to be stable after the positive edge of the clk.

Example:



Single-Cycle Datapath and Control

	BrEq	BrLt	PCSel	ImmSel	BrUn	ASel	BSel	ALUSel	MemRW	RegWEn	WBSel
add	*	*	+4	*	*	Reg	Reg	Add	Read	1	ALU
sub	*	*	+4	*	*	Reg	Reg	Sub	Read	1	ALU
addi	*	*	+4	I	*	Reg	Imm	Add	Read	1	ALU
lw	*	*	+4	I	*	Reg	Imm	Add	Read	1	Mem
sw	*	*	+4	S	*	Reg	Imm	Add	Write	0	*
beq	0	*	+4	B	*	PC	Imm	Add	Read	0	*
beq	1	*	ALU	B	*	PC	Imm	Add	Read	0	*
bne	0	*	ALU	B	*	PC	Imm	Add	Read	0	*
bne	1	*	+4	B	*	PC	Imm	Add	Read	0	*
jalr	*	*	ALU	I	*	Reg	Imm	Add	Read	1	PC+4
jal	*	*	ALU	J	*	PC	Imm	Add	Read	1	PC+4

Open MPI

```
int main (int argc, char ** argv) {

    int numtasks = atoi(argv[1]); //read n from cmd line

    MPI_Init(&argc, &argv); //initialize

    int procID, totalProcs;

    MPI_Comm_Size(MPI_COMM_WORLD, &totalProcs);   } get process ID of this
    MPI_Comm_Rank(MPI_COMM_WORLD, &procID);       } process and total # of processes

    if (procID == 0) {  // are we manager or worker?

        // manager  ------>  int nextTask = 0;  // next task to do
                              MPI_Status status;
                              int32_t message;
                              while(next_Task < numTask) {  // assign tasks
                                  MPI_Recv(...); // wait for a message from any worker
                                  int sourceProc = status.MPI_Source; // process ID of the source of msg
                                  message = nextTask; // assign next task
                                  MPI_Send(...);
                                  nextTask++;
                              }
                              for (int i=0; i < TotalProcs-1; i++) { // wait for all processes to finish
                                  MPI_Recv(...); // wait for a msg from any worker
                                  int sourceProc = status.MPI_SOURCE // process ID of the source of msg
                                  message = TERMINATE;
                                  MPI_Send(...)
                              }
    } else {

        // worker  ------>  int32_t message;

                            while(true) {
                                //request more work
                                message = READY;

                                MPI_Send(...);
                                // recieve message from manager
                                MPI_Recv(..);
                                if(message == TERMINATE) break; // all done!
                                matmul(message);
                            }

    }

    MPI_Finalize();
}
```