CS61C Course Notes

Anmol Parande

Fall 2019 - Professors Dan Garcia and Miki Lustig

Contents

Disclaimer: These notes reflect 61C when I took the course (Fall 2019). They may not accurately reflect current course content, so use at your own risk. If you find any typos, errors, etc, please report them on the GitHub repository.

1 Binary Representation

Each bit of information can either be 1 or 0. As a result, N bits can represent at most 2^N values.

1.1 Base conversions

Binary to Hex conversion

- Left pad the number with 0's to make 4 bit groups
- Convert each group its appropriate hex representation

Hex to Binary Conversion

- Expand each digit to its binary representation
- Drop any leading zeros

1.2 Numeric Representations

When binary bit patterns are used to represent numbers, there are nuances to how the resulting representations are used.

Definition 1 *Unsigned integers are decimal integers directly converted into binary. They cannot be negative.*

Definition 2 *Signed integers are decimal integers whose representation in binary contains information denoting negative numbers.*

Binary math follows the same algorithms as decimal math. However, because computers only have a finite number of bits allocated to each number, if an operation's result exceeds the number of allocated bits, the leftmost bits are lost. This is called **overflow**

1.2.1 Sign and Magnitude

In the sign and magnitude representation, the leftmost bit is the sign bit. A 1 means the number is negative where as 0 means it is positive. The downside to this is when overflow occurs, adding one does not wrap the bits properly.

1.2.2 One's Complement

To fix the bit wrapping, to form the negative number, we can flip each bit. This solves the wrapping issue so even with overflow, when adding 1, the result will continue increasing.

1.2.3 Two's Complement

To convert from decimal to two's complement: If the number is positive, convert to binary as normal. If the number is Negative:

1. Convert the positive version of the number to binary
2. Invert each bit
3. Add one to the result

If given a two's complement number, you can find -1 times that number by following the same process (invert bits and add 1).

1.2.4 Bias Encoding

With bias encoding, the number is equal to its unsigned representation plus a bias turn. With a negative bias, we can center a positive range of $0 \to 2^N - 1$ on 0.

1.3 Floating Point Representation

To represent decimal numbers in binary, we use floating point representation. Any number in scientific notation has the following components

- Mantissa: The number in front of the point
- Significand: The digits after the point
- Radix: The base of the number
- Exponent: How many times the point should be shifted to recover the original number

$$\underbrace{1}_{\text{Mantissa}} \underbrace{.}_{\text{Binary Point}} \underbrace{01}_{\text{significand}} \cdot \underbrace{2^{\overbrace{-1}^{\text{exponent}}}}_{\text{radix}}$$

If we have a 32 bit system, then by the floating point convention:

$$\underbrace{0}_{\text{Sign Bit}} \; \overbrace{00000000}^{\text{Exponent Bits}} \; \overset{\text{Significand}}{\underbrace{00000000000000000000000}}$$

In the floating point representation, the manissa is always one because we are using scientific notation, so we don't bother storing it.

- Sign Bit: Determines the sign of the floating point number
- Exponent: 8 bit biased number (-127 bias)
- Significand: 23 significand bits representing $2^{-1}, 2^{-2}$...

Given a number in floating point representation, we can convert it back to decimal by applying the following formula:

$$n = (-1)^s(1 + significand) \cdot 2^{exponent-127}$$

Notice a couple things about this representation.

- If the exponent is larger than 8 bits, then overflow will occur
- If a negative exponent is more than 8 bits, then underflow occurs
- There are 2 0's. Positive 0 and negative 0

Because of the way that floating point is built, certain sequences are designated to be specific numbers

Exponent	Significand	Object
0	0	0
0	nonzero	denorm
1-254	anything	± #
255	0	$\pm\infty$
255	nonzero	NaN

A **denormed number** is a number where we don't have an implicit 1 as the mantissa. These numbers let us represent incredibly small numbers. They have an implicit exponent of -126.

2 C

2.1 C Features

- C is a compiled language $\implies$ executables are rebuilt for each new system
- Every variable holds garbage until initialization
- Function parameters are pass by value

2.2 Bitwise Operators

Bitwise operators are operators which change the bits of integer-like objects (ints, chars, etc)

- &: Bitwise AND. Useful for creating masks
- |: Bitwise OR. Useful for flipping bits on
- $\wedge$: Bitwise XOR. Useful for flipping bits off
- <<: Left shifts the bits of the first operand left by the number of bits specified by the second operand.
- >>: Right shifts the bits of the first operand right by the number of bits specified by the second operand.
- $\sim$: Inverts the bits.

2.3 Pointers

Definition 3 *The address of an object is its location in memory*

Definition 4 *A pointer is a variable whose value is the address of an object in memory*

2.3.1 Pointer Operators

- &: Get the address of a variable
- $*$: Get the value pointed to by a pointer

Because C is pass by value, pointers allow us to pass around objects without having them copied. They can also lead to cleaner, more compact code. However, always remember that declaring a pointer creates space for an object but **does not** put anything in that space (i.e it will be garbage).

Pointers can point to anything, even other points. For example, $int**a$ is a pointer to an integer pointer. This is called a handle

2.4 Arrays

In C, arrays are represented as adjacent blocks of memory. The way that we interact with them is through pointers. Consider

```
int a[];
int *b;
```

Both of these variables represent arrays in C. They point to the first memory location of the array. C arrays can be indexed into using [] subscripts like other programming languages. They can also be index into using pointer arithmatic. For example, $*(a+2) \equiv a[2]$. By adding 2 to a, C knows to look two memory locations into the array (i.e the third element).

C is able to do this because it automatically computes the size of the objects in the array to know how much to advance the pointer by.

```
int a[] = {1, 2, 3, 4, 5}; //ints are 4 bytes;
printf("%u", &a); // 0x2000
printf("%u", &(a+2)); // 0x2008
char *c = "abcdef"; //chars are 1 byte
printf("%u", &c); // 0x3000
printf("%u", &(a+c)); // 0x3002
```

One important thing to remember is that declared arrays are only allocated while the scope is valid. That means once a function returns, their memory is free to be taken.

Another important consideration in C is that C arrays do not know their own lengths and they do not check their bounds. This means whenever you are passing an array to a function, you should always be passing its size as well.

2.5 Structs

Structs are the basic datastructures in C. Like classes, they are composed of simpler data structures, but there is no inheritance.

2.5.1 Struct Operators

- $->$: dereference a struct and get a subfield

typedef can be a useful command with structs because it lets us name them cleanly.

3 Memory Management

3.1 Memory Basics

In memory, a **word** is 4 bytes. When objects are saved to memory, they are saved by words. How the words are ordered depends on the type of system. In **Little Endian** systems, the Least Significant Byte is placed at the lowest memory address. In other words, the memory address points to the least significant byte

The opposite is true in **Big Endian** systems. For example, lets say we have the number $0x12345678$ stored at the memory address $0x00$

	$0x00$	$0x04$	$0x08$	$0x0C$
Little Endian:	78	56	34	12
Big Endian:	12	34	56	78

There are four sections of memory: the stack, the heap, static, and code.

Definition 5 *The Stack is where local variables are stored. This includes parameters and return addresses. It is the "highest" level of memory and grows "downward" towards heap memory.*

Definition 6 *The Heap is where dynamic memory is stored. Data lives here until the programmer deallocates it. It sits below the stack and grows upwards to toward it.*

Definition 7 *Static storage is where global variables are stored. This storage does not change sizes and is mostly permanent.*

Definition 8 *Code storage is where the "code" is located. This includes preprocessing instructions and function calls.*

Definition 9 *Stackoverflow is when the stack grows so large that it intersects with heap memory. This is mostly unavoidable.*

Definition 10 *Heap pileup is when the heap grows so large that it starts to intersect with the stack. This is very avoidable because the programmer manages it.*

Definition 11 *All of the memory which a program uses is collectively referred to as the address space of the program.*

3.2 The Stack

The stack is named that way because every time a function call is made, a stack frame is created. A stack frame includes the address of the return instruction and the parameters to the function. As the function executes, local variables are added to the frame. When the function returns, the frame is popped off. In this way, frames are handled in Last-In-First-Out (LIFO) order.

Definition 12 *The stack pointer is a pointer which points to the current stack frame.*

Important: Deallocated memory is not cleared. It is merely overwritten later.

3.3 The Heap

The heap is a larger pool of memory than the stack and it is not in contiguous order. Back to back allocations to the heap may be very far apart.

Definition 13 *Heap fragmentation is when most of the free memory in the heap is in many small chunks*

Fragmentation is bad because if we want to allocate space for a large object, we may have enough cumulative space on the heap, but if none of the remaining contiguous spaces are open, then there is no way to create our object.

Implementation:

Every block in the heap has a header containing its size and a pointer to the next block. The free blocks of memory are stored as a circular linked list. When memory needs to be allocated to the heap, this linked list is searched. When memory is freed, adjacent empty blocks are coalesced into a single larger block. There are three different strategies which can be used to do this allocation/freeing.

Definition 14 *Best-fit allocation is where the entire linked list is searched to find the smallest block large enough to fit the requirements*

Definition 15 *First-fit allocation is where the first block that is large enough to fit the requirement is returned*

Definition 16 *Next-fit allocation is like first-fit allocation except the memory manager remembers where it left off in the free list and resumes searching from there.*

3.4 Heap Management

As a C programmer, it is up to us to manage the heap memory. This is done through the *malloc* function.

```
malloc(size_t size)
```

Malloc takes a size in bytes and returns a pointer to the allocated memory in the heap. If there is no space in the heap, then it returns $NULL$. Any space created with *malloc* must be freed using the $free$ function.
Things to watch out for:

- Dangling reference (A pointer used before malloc)
- Memory Leak (When $free$ isn't called or the pointer is lost but $free$ wasn't called)
- Do not free the same memory twice
- Do not use free on something not created with malloc
- malloc does not overwrite what was in the current memory location.

Because we need to tell *malloc* how much space we need allocated, we will often use the $sizeof$ function. This returns the size in bytes of whatever type or object you give it. This allows us to write code for different architectures (i.e 32bit vs 64bit)

4 RISC-V

4.1 Basics of Assembly Languages

In Assembly Languages like RISC-V, instructions are executed directly by the CPU. The basic job of the CPU is to taken a set of instructions and execute them in order. The **Instruction Set Architecture** (ISA) determines how this is done. In **Reduced Instruction Set Computing** (RISC) languages, the instruction set is limited because it makes the hardware simple. Complex operations are left to the software.

Registers are the operands of assembly language operations. A register is a memory location on the CPU itself. There are a limited number of them, but they are very fast. In RISC-V processors, there are 32 registers. Each register can store one **word**. This is 32 bits on a 32-bit system.

Registers are labeled x0-x31. x0 is a special register because it always holds the value 0. Unlike variables, registers have no types. The operation is what determines what the content of the register is treated as.

Immediates are numerical constants. They can also be used as the operands of assembly intructions.

4.2 RISC-V Structure

The general format of a RISC-V instruction is

```
operation_name destination source1 source2
```

Labels are text in the program which denote certain locations in code. **Branches** change the flow of the program, usually by jumping to a label or an address in the code portion of memory.

Pseudo-instructions are instructions which are translated into different instructions by the assembler. They exist because they increase readability of the program.

In order to increase program legibility, labels are hardly referred to by their number (x15, x20, etc). Instead, they have symbolic names. Here are a few.

- a0 - a7: The argument registers
- s0 - s7: The saved registers
- ra: return address register
- sp: stack pointer register
- pc: Program Counter

4.2.1 Caller Callee Convention

Because functions can always overwrite registers, programmers set up conventions for calling and returning from functions. The **Caller** is the function which calls another function. The **Callee** is a function which is being called.

Functional Control Flow

1. Put paramters where the function can access them (a0-a7)
2. Transfer control to the function (jump)
3. Acquire the local storage resources for function (Increase stack)
4. Perform the desired task of the function
5. Put result in $a0$ where the calling function can access it

6. Release local variables and return data to used registers so the caller can access them (Decrease stack)
7. Return control to the calling function (Jump to ra)

Because every function must have this control flow, the caller-callee convention was set up as follows

1. sp, gp, tp, s0-s11 are preserved across a function call
2. t0-t7, a0-a7 are not preserved across a function call

If a register is not preserved across a function call, then the caller cannot expect its value to be the same after the callee returns. In order to preserve registers across a function call, we use the **stack**. The Stack Frame stores the variables which need to be saved in order to adhere to caller callee convention. Every RISC-V function has a **Prologue** where it saves the neccessary registers to the stack. An example might look like

```
addi sp, sp, -16
sw s0, 0(sp)
sw s1, 4(sp)
sw s2, 8(sp)
sw ra, 12(sp)
```

This function must use the s0-s3 saved registers. We first create the stack frame by decrementing the stack pointer. Then we save the saved registers to the newly allocated memory. This function must be calling another function, so it has to remember its return address. That is why we save it to stack. The **Epilogue** is the part of the function before it returns where everything on the stack is put back.

```
lw s0, 0(sp)
lw s1, 4(sp)
lw s2, 8(sp)
lw ra, 12(sp)
jr ra
```

4.2.2 Directives

Directives are special demarcations in an assembly file which designated different pieces of data.

.text: Subsequent items are put into the text segment of memory (i.e the code)

.data: Subsequent items are put into the data segment of memory (i.e static variables)

.global sym: Declares a symbol to be global, meaning it can be referenced from other files

.string str Stores a null-terminated string in the data memory segment

.word Store n 32 bit quantities into contiguous memory words.

4.3 Instruction Formats

In a **Stored Program Computer**, instructions are represented as bit patterns. They are stored in memory just like data. As a result, everything has a memory address, including lines of code. The **Program Counter** is the memory address of the current instruction. Each instruction in RISC-V is 1 word, or 32 bits. It is divided into fields, each of which tells the processor something about the instruction.

R Type:	Register-register arithmetic
I Type:	Register-immediate arithmetic
S Type:	Store instructions
B Type:	Branch instructions
U Type:	20 bit upper immediate instructions
J Type:	Jump Instructions

Every instruction type has a 7 bit **opcode** which tells the processor what instruction type it is processing. These are always the last 7 bits. Some instructions also have **funct3** and **funct7** fields which define the operation to perform in conjunction with the opcode.

R-Type

31:25	24:20	19:15	14:12	11:7	6:0
funct7	rs2	rs1	funct3	rd	opcode

I-Type

31:20	19:15	14:12	11:7	6:0
Imm[11:0]	rs1	funct3	rd	opcode

S-Type

31:25	24:20	19:15	14:12	11:7	6:0
Imm[11:5]	rs2	rs1	funct3	Imm[4:0]	opcode

B-Type

31	30:25	24:20	19:15	14:12	11:8	7	6:0
Imm[12]	Imm[10:5]	rs2	rs1	funct3	Imm[4:1]	Imm[11]	opcode

U-Type

31:12	11:7	6:0
Imm[31:12]	rd	opcode

J-Type

31	30:21	20	19:12	11:7	6:0
Imm[20]	Imm[10:1]	Imm[11]	Imm[19:12]	rd	opcode

4.3.1 Addressing

Notice that the J-Type and B-Type instructions require use a label in code. These labels are encoded in the instruction format as an offset from the program counter. This is known as **PC Relative Addressing**. Since each instruction in RISC-V is 1 word, we will never have an odd address. As a result, we don't need to store the last bit of the immediate in B-Type and J-Type instructions because it is automatically 0.

4.4 Compiling, Assembling, Linking, Loading

Compiling, Asssembling, Linking, and Loading (**CALL**) are the four steps of loading a program.

4.4.1 Compiler

The input to the compiler is a file written in a high level programming language. It outputs assembly language code which is built for the machine the code was compiled on. The output of the compile may still include pseudo-instructions in assembly.

4.4.2 Assembler

The Assembler is the program which converts assembly language code to machine language code. It reads and uses directives, replaces psuedo-instructions with their real equivalent, and produces machine language code (i.e bits) where it can.

The output of the assembler is an object file. The object file contains the following elements:

- Object file header: The Size and position of the different sections of the object files
- Text Segment: The code
- Data Segment: binary representation of the static data in the source
- Relocation Table: A special data structure which contains the lines of code needing to be fixed

- Symbol Table: A special data structure which lists the files global labels and static data labels
- Debugging information

The symbol table contains information which is public to other files in the program.

- Global function labels
- Data Labels

The Relocation Table contains information that needs to be relocated in later steps. It essentially tracks everything that the Assembler cannot directly convert to machine code immediately because it doesn't have enough information.

- List of labels this file doesn't know about
- List of absolute labels that are jumped took
- Any piece of data in the static section

When the assembler parses a file, instructions which don't have a label are converted into machine language. When a label is defined, it's position is stored in the relocation table. When a label is encountered in code, the assembler looks to see if it's position was defined in the relocation table. If it is found, the label is replaced with the immediate and converted to machine code. Otherwise, the line is marked for relocation.

In order to do its job, the assembler must take two passes through the code. This is because of the **forward reference problem**. If a label is used before it is defined in the same file, the first time the assembler encounters it, it won't know how to convert it to machine code. To resolve this, the assembler simply takes two passes so it finds all labels in the first pass and convert the lines it originally couldn't in the second pass.

4.5 Linker

The Linker is responsible for linking all of the object files together and producing the final executable. The linker operates in three steps:

1. Put the text segments together
2. Put the data segments together
3. Resolve any referencing issues

After the linking step, the entire program is finally in pure machine code because all references to labels must be resolved. The linker knows the length of each text and data segment. It can use these lengths to order the segments appropriately. It assumes that the first word will be stored at $0x10000000$ and can calculate the absolute address of each word from there. To resolve references, it uses the relocation table to change addresses to their appropriate values.

- PC-Relative Addresses: Never relocated
- Absolute Function Addresses: Always relocated
- External Function Addresses: Always relocated
- Static data: Always relocated

When the linker encounters a label, it does the following.

1. Search for the reference in the symbol tables
2. Search for the reference libraries if the reference is not in the symbol tables
3. Fill in the machine code once the absolute address is determined

This approach is known as **Static Linking** because the executable doesn't change. By contrast, with dynamic linking, libraries are loaded during runtime to make the program smaller. However, this adds runtime overhead because libraries must be searched for during runtime.

4.6 Loader

The Loader is responsible for taking an executable and running it.

1. Load text and data segments into memory
2. Copy command line arguments into stack
3. Initialize the registers
4. Set the PC and run

5 Synchronous Digital Systems

Synchronous Digital Systems are the systems which run in our computers. They are **Digital** because electric signals are interpreted as either a 1(asserted) or a 0 (unasserted). They are **Synchronous** because all of the operations are coordinated by a clock. The clock is a device in the computer which alternates between being 1 and 0 at a regular interval, called the **clock period**. There are two critical elements in SDS.

1. Combinational logic element: An element whose output is only a function of the inputs
2. State Element: An element which stores a value for an indeterminate amout of time

Combinational logic elements are used to perform operations on inputs while state elements are used to store information and control the flow of information between combinational logic blocks.

5.1 Registers

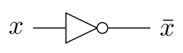
Registers are state elements frequently used in circuits. On the rising edge of the clock, the input d is sampled and transferred the output q. At all other times, the input d is ignored. There are three critical timing requirements which are specific to registers

1. Setup Time: How long the input must be stable before the rising edge of the clock for the register to properly read it
2. Hold Time: How long the input must be stable after the rising edge of the clock for the register to properly read it
3. Clock-to-Q Time: How long after the rising edge of the clock it takes for input to appear at the registers output

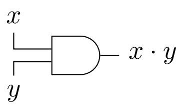
5.1.1 Pipelining

One place where registerrs become useful is in pipelining. Pipelining a circuit means placing registers after combinational logic blocks. This stops delay times from adding up because now intermediate quantities must be stored in the register before being passed to the next block. This allows for higher clock frequencies because we are no longer limited by the logic delays, only by our registers.

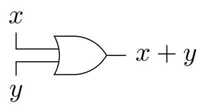
5.2 Combinational Logic Elements



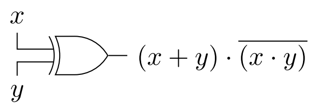
Not Gate



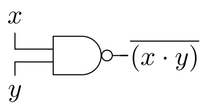
And Gate



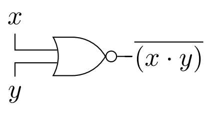
Or Gate



Xor Gate



Nand Gate



Nor Gate

These are the basic gates which are used in combinational logic. Using Boolean Algebra, we can simplify complicated logic statements/circuits.

$$\begin{array}{c|c}
x \cdot \bar{x} = 0 & x + \bar{x} = 1 \\
x \cdot 0 = 0 & x + 1 = 1 \\
x \cdot 1 = x & x + 0 = x \\
x \cdot x = x & x + x = x \\
x \cdot y = y \cdot x & x + y = y + x \\
(xy)z = x(yz) & (x + y) + z = x + (y + z) \\
x(y + z) = xy + xz & x + yz = (x + y)(x + z) \\
xy + x = x & (x + y)x = x \\
\overline{xy} = \bar{x} + \bar{y} & \overline{x + y} = \bar{x}\bar{y}
\end{array}$$

One other important circuit element is the data multiplexor (mux). A mux takes in n different sources of data and selects one of those sources to output. It does this using select inputs. For example, if a mux has 3 select bits, then it can choose from 8 different streams of data.

5.3 Timing

In circuits, timing is incredibly importat because combinational logic blocks have propagation delays (i.e the output does not change instantaneously with the input). When used in conjuction with registers (which what their own timing requirements), if one is not careful, we could build a circuit which produces inaccurate outputs. When checking whether or not a circuit will do what it is designed to do, we need to look at two special paths:

- Longest CL Path: The longest path of combinational logic blocks between two registers
- Shortest CL Path: The shortest path of combinational logic blocks between two registers.

This will let us analyze things like the maximum clock rate we can achieve or the maximum hold time our registers can have.

$$t_{delay} = t_{setup} + t_{clk2q} + t_{longestCL}$$

This first formula tells us the maximum amount of time it takes for an input to propagate from a register to another register. When the clock rises, the first register will read the input and output it t_{clk2q} later. The input will then go through the combinational logic elements. Since we are looking for the maximum, we only consider the longest CL path. Finally, the output needs to be stable for t_{setup} in order for the destination register to read it properly. As long as our clock period is longer than the max delay, our circuit will work properly.

$$t_{hold} = t_{clk2q} + t_{shortestCL}$$

This second formula tells us the maximum hold time which our registers can have. When the clock rises, the first register reads the input and outputs it t_{clk2q} later. The input will then go through the combinational logic elements.

Hold time is how long the input needs to be stable after the rising edge of the clock, so if the combinational logic works incredibly fast, we will need a fast hold time because otherwise the output will change faster than the hold time of the output register. This is why we look at the shortest CL path. As long as our registers have a hold time which is less than the max-hold, our circuit will work properly.

6 Datapath

The processor is the part of the computer which manipulates data and makes decisions. The processor is split into two parts: datapath and control. The datapath is the part of the processor with the necessary hardware to perform the operations required by the ISA. THe control tells the datapath what needs to be done. On every tick of the clock, processors execute a single instruction. Instruction execution takes place in 5 stages.

1. Instruction Fetch
2. Instruction Decode
3. Execute
4. Memory Access
5. Write Back

Each datapath has several main state elements

- Register File: An array of registers which the processor uses to keep information out of memory
- Program Counter: A special register which keeps track of where the processor is in the program
- IMem: A read-only section of memory containing the instructions that need to be executed
- DMem: Section of memory which contains data the processor needs to read/write to

Important combinational logic elements of the datapath include

- Immediate Generator: Generates the immediate
- Branch Comparator: Decides whether or not branches should be taken
- ALU: Performs mathematical operations

All parts of the datapath operate at the same time. The control is how the processor chooses which output to work with. Control can either be done through combinational logic or by using ROM (Read-Only Memory).

6.1 Pipelined Datapath

A single-cycle datapath is one where every instruction passes through the datapath one at a time. However, this is inefficient because faster stages (such as register reading) are left unused while waiting for slower stages (such as memory reading). One way to fix this is to pipeline the datapath. This allows multiple instructions to use different parts of the datapath at once, speeding up the processor because no stage is left unutilized. All we need to do is add registers after each datapath stage. However, this introduces "hazards" into the processor.

6.1.1 Structural Hazards

Definition 17 *A structural hazard is when two or more instructions compete for access to the same physical resource.*

One solution is to have the instructions take turns to access the resource. The other option is to add more hardware to distribute that resource. For example, a Regfile structural hazard would be when the processor needs to read registers for one instruction and write a register for another. This is solved by giving the Regfile two independent read ports and one independent write port. Another example is a memory structural hazard where instructional memory and data memory are used simultaneously. This is solved by separating them into IMem and DMem.

6.1.2 Data Hazards

Definition 18 *A data hazard is when two instructions have a data dependency between them*

Problem: One instruction is reading from a register that a previous instruction is writing back to
Solution: The WB stage will always update the value first before Instruction Decode reads a new value

Problem: The result from the ALU will take 2 cycles to be written back. An instruction might need it before then
Solution 1: We could stall by introducing no-ops (instructions that do nothing), but this would kill performance.
Solution 2: WE can add a loop from the output of the ALU to the ALU input via a mux and add a control element to determine when we should use the forwarded value

Problem: Suppose a load instruction is followed by an instrucction that requires the data loaded from DMEM. The ALU thus needs the data as it is being read. This slot after a load is known as the load delay slot. If the load delay slot instruction uses the result of the load, then we need to stall for a cycle
Solution 1: We could insert a no-op into the code

Solution 2: Turn off all write enables and run the second instruction twice
Solution 3: Reorder instructions so that the load delay slot doesn't use the loaded result

6.1.3 Control Hazards

Definition 19 *A control hazard is when the program flow changes so instructions in the pipeline are no longer relevant*

Notice that this is only a problem when a branch is taken because it means the pipeline must be flushed to get the wrong instructions out of the pipeline (by converting them to no-ops). A more advanced way to fix this is to implement branch prediction which is where the processor attempts to predict whether or not a branch is taken and load instructions based on that.

7 The Memory Hierarchy

Computers have several levels of memory, each one serving different purposes. On the processor itself, we have the Regfile which provides fast read and write access. Right below it, we have the caches as welll as main memory (a.k.a DRAM) which has more capacity than registers but is much slower. After DRAM, we have Disk, which is even larger but incredibly slow. Different systems control the transfer of data between different parts of memory.

- Registers $\leftrightarrow$ Memory: The compiler/assembly programmer
- Cache $\leftrightarrow$ Main Memory: Cache Controller
- Main Memory $\leftrightarrow$ Disk: Operating System

7.1 Caching

The closer memory is to the processor, the faster it is. However, that also makes it more expensive. The idea of caching is to copy a subset of main memory and keep it close to the processor. We can then stack multiple layers of caches (each one with more storage than the other) before we actually need to read DRAM.

Definition 20 *Temporal locality is the idea that if data is used now, it will be used later on*

Definition 21 *Spatial locality is the idea that if data is used now, nearby data will likely be used later*

Caches take advantage of both temporal and spatial locality to provide quick memory access.

7.1.1 Direct Mapped Cache

In a direct mapped cache, data is transferred between main memory and the cache in segments called blocks. Each memory address is associated with one possible block in the cache. That way, when checking whether or not the data located at a particular address is located in the cache, we only need to check a single location. Each address is split into three parts.

- Offset: The number of the byte within the block that should be accessed.
- Index: The block in the cache
- Tag: Remaining bits which distinguish all memory addresses mapping to the same block

Tag	Index	Offset

In this way, the direct-mapped cache is similar to a Hashmap where the key is the Index and the Tag. When given an address, the direct mapped cache will first go the block specified by the address' index. It will then compare the tags.

Important: Note that in addition to the data stored in memory, the DM Cache also stores the tag that data is associated with.

7.1.2 Fully Associative Cache

In a fully associated cache, the cache is divided into a set of blocks. These blocks contain a tag and the data stored inside memory. When checking whether or not a particular memory address is inside the cache, we check all of the tags in parallel. Because there is no concept of positioning within the cache, we don't need an index, so each address is split into 2 parts.

- Offset: the number of the byte within the block that should be accessed
- Tag: Remaining bits which distinguish memory addresses from each other

Tag	Offset

In this way, fully associative caches are like an array. When looking data up in the array, we search through it. The benefit of this is that we have no more conflict misses because there are no mappings between memory addresses. The drawback is that we need a comparator for every single entry.

7.1.3 N-Way Set Associative Cache

An N-Way set associative cache combines the ideas of a direct mapped caches and fully associative caches. Like the direct mapped cache, the N-Way set is divided into a series of sections called sets. Each set is fully associative cache with N blocks inside. When checking whether or not the data for a particular memory address is stored inside the cache, we first identify the set using the index and then check all of the tags in that set. This is why the address is split into 3 parts:

- Offset: The number of the byte within the block that should be accessed.
- Index: The set the data maps to
- Tag: Remaining bits which distinguish all memory addresses mapping to the same block

Tag	Index	Offset

We can think of N-Way Fully Associative caches like Hashmaps where the key is the index and the value is an array. We must search through the array (by checking tags) to actually get the correct block and then use the offset to retrieve the data at a location in the block. N-Way fully associative caches are useful because now we get the benefits of fully-associative caches but we only need N comparators to check the tags, allowing us to build larger caches.

7.1.4 Cache Metrics

The general flow of cache access is as follows:

1. The processor issues the address to the cache
2. If the data is in the cache, the data is sent to the processor
3. If the data is not in the cache, memory will read the address, load the data into the cache, save the cache, and then the cache sends the data to the processor

Definition 22 *A cache hit is when the cache contains the requested data and passes it to the processor*

Definition 23 *A cache miss is when the cache does not contain the requested data, so it must read from memory*

There are 3 different types of cache misses

- Compulsory Miss: When the program first starts, the cache is empty so misses are guaranteed.
- Conflict Miss: Two memory addresses map to the same cache location so the one currently in the cache must be replaced
- Capacity Miss: Cache capacity is full so something must be replaced (this is the primary miss of a associative cache)

The hit rate is the fraction of accesses which are hits whereas the miss rate is the fraction of accesses that are misses. It takes time to replace data in the cache because it must be read from memory, so there is some miss penalty which is how long it takes to replace that data. Likewise, the hit time is how long it takes to access cache memory.

When choosing parameters for a cache, one must keep these metrics in mind. For example, a large block size might give us better spatial locality, but eventually the number of cache misses will increase, and they will have a larger penalty because more memory is being accessed at once.

7.1.5 Writing to the Cache

Reading data from the cache is simple because data just needs to be passed to a single location (the processor). However, writing data means the value must be updated in both memory and the cache. Caches have a write policy which they use to determine how they handle writes.

Definition 24 *A write-through cache policy is when writes update both the cache and the memory*

Definition 25 *A write-back cache policy is when a write updates data in the cache and only updates memory once the data is removed from the cache.*

A write-back policy is more complicated because it allows memory to be stale for some time. To keep track of which blocks in the cache have been written to, we add a dirty bit to the cache and update memory if a dirty block is removed from the cache. This allows the memory and cache to be inconsistent but can be useful if we are making multiple writes at the same time because we don't have to go to memory each time.

7.1.6 Block Replacement

Just like caches can have different write policies, they can also have different replacement policies which determine how to replace a block if there is a cache miss. For a Direct mapped cache, the index fully specifies the location of the block, so the only possible policy is a whole block replacement of whatever was already in that location. For N-Way and Fully Associative caches, however, we have a choice as to which block in a set we want to replace.

Definition 26 *A First in First Out (FIFO) replacement policy will remove the block that entered the set first*

A FIFO policy doesn't take into account accesses. Because of this, even frequently accessed data can be removed from the cache.

Definition 27 *A Least Recently Used (LRU) replacement policy will remove the block from the set which has been accessed the least recently.*

LRU replacement policies make sure that frequently accessed data stays in the cache. Note that both FIFO and LRU replacement policies only kick in once the Fully Associative cache is full or a set in the N-Way Associative cache is full.

7.2 Virtual Memory

So far, we have been assuming our computer is only running a single program at a time, so each program has access to the full memory. This, however, is never the case with modern computers. Modern computers run hundreds of processes simultaneously. At an even more basic level, not all computers have the same size of main memory, so how can memory-intensive programs run on low-RAM machines? These are the problems which virtual memory tackles.

Definition 28 *An address space is the set of addresses for all available memory locations.*

The main idea of **Virtual Memory** is to have programs operate using virtual addresses and have the operating system translate those into physical addresses.

Definition 29 *The virtual address space is the set of all addresses that the user program knows about and has available to run its code.*

Definition 30 *The physical address space is the set of addresses which map to a location in physical memory (i.e DRAM)*

Both the virtual address space and physical address space are chunked into units called pages (similar to blocks in caches). Pages are the units of transfer from Virtual Memory to Physical Memory and Physical Memory to Disk.

Definition 31 *The page table tells us which virtual page maps to which physical page*

The page table also contains information such as access rights for that page.

Definition 32 *The page table base register tells the processor where the page table is located in memory.*

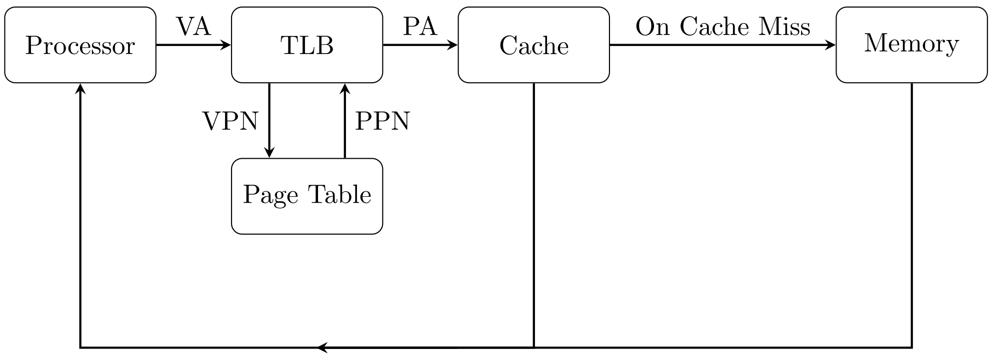
Because we must go to memory for the page table, every single memory access would take 2 trips to memory (one for the page table, the other for the data). To stop this, we use a **Translation Lookaside Buffer**(TLB) to cache the page table entries so we can still access them quickly.

7.2.1 Memory Access

Each virtual address is split into two parts: the virtual page number (VPN) and the page offset.

VPN	Page Offset

The VPN is further divided into the TLB tag and TLB index. We first check the TLB at the TLB index if the tags match. If they do, then the TLB will return the PPN. If not, we need to check the page table for the mapping between VPN and PPN. Once the PPN is retrieved, we concatenate the PPN with the page offset. This gives us a physical address which we can then use to check the cache and main memory.



7.2.2 Virtual Memory and the Disk

Suppose that a program has acccess to a very large address space, but the machine only has a small amount of physical memory. In that case, not all pages in virtual memory can correspond to a page in physical memory. The solution to this is to save pages to disk when they are not being used (called swap space).

Definition 33 *A page fault is when a page needs to be accessed but it is in disk and not physical memory*

When a page fault occurs, there is no entry in the page table which corresponds to a VPN, so the OS will update the page table by loading the page from disk into memory. Since this takes a long time, frequently it will switch to a different process while the page is being loaded. When a system is constantly swapping pages from disk to memory, this is called thrashing.

7.2.3 Virtual Memory Performance Metrics

Definition 34 *The TLB miss rate is the fraction of TLB accesses resulting in a TLB miss*

Definition 35 *The Page Table Miss Rate is the fraction of page table accesses that result in a page fault*

8 Parallelism

Parallelism is the concept of doing multiple things at the same time. This can be done in both hardware and software. We classify parallelism based on

Flynn's taxonomy which breaks things down into how a program handles its data streams and how the program handles instruction streams.

Definition 36 *Single Instruction, Single Data (SISD) programs are normal programs where instructions are executed sequentially and there is a single source of data*

Definition 37 *Single Instruction, Multiple Data (SIMD) programs are like SISD in that they take a single sequence of instructions, but they leverage special instruction which apply the same operation to many pieces of data at a time (i.e vector operations)*

Definition 38 *Multiple Instruction, Multiple Data (MIMD) programs run instructions simultaneously and like SIMD, they use instructions which operate on many data at a time*

The fourth part of the taxonomy is Multiple Instruction, Single Data (MISD), but it is not as commonly used.

8.1 Data Level Parallelism and SIMD

The basic idea of data level parallelism is to execute one operation on multiple data streams at a time. To do so, we use special registers like the XMM registers which are designed for floating point operations. These registers allow us to pack together data into a single register and perform an operation on the entire contents of the register.

8.2 Thread Level Parallelism

8.2.1 Threads

Definition 39 *A thread is a sequential flow of instructions that are performing some task*

Each thread has

- A Dedicated Program Counter
- Registers
- Access to Shared Memory

Each core in a multiprocessor has multiple hardware threads which can execute instructions. The Operating System multiplexes software threads onto these hardware threads. All the threads which are not mapped to hardware threads are marked as waiting. The OS chooses which threads run by

1. Remove a blocked/finished thread by interrupting execution and saving the registers/PC

2. Multiplexing a new software thread onto the hardware thread by loading the registers and PC
3. Starting execution by jumping to the saved PC and running the processor

8.2.2 Synchronization

Thread-Level Parallelism introduces some complications, however, because multiple processors have to work together to make the program work. One issue which can arise synchronization. Suppose two threads need to update the same variable. We don't know which order the threads will access the variable in, so it is possible they could completely overwrite each other by mistake. To solve this proble, we use locks.

Definition 40 *A lock (or semaphore) is a variable which stops other threads from updating a variable until the thread holding the lock releases it.*

In RISC-V, atomic memory operations (AMO) were made to handle these synchronization issues in hardware. AMO operations perform an operation on an operand in memory and set the destination register to the original memory value. This allows for a read and write in a single instruction. Locks can be dangerous however. Sometimes, the system can reach a **deadlock** where no progress is made because all threads are waiting on each other in a circular fashion.

8.2.3 Cache Coherence

When there are multiple cores to a machine, they all have access to the same shared memory (DRAM). However, each core has its own set of caches. This means that it is possible for the cache of one core to out of date because a different core changed a shared variable. To make sure our caches are coherent

- When a processor has a cache miss or write, other processors are notified
- Writes by one processor are snooped by the other processors and they invalidate their caches

To help manage coherence, we add a couple more bits to the cache in addition to valid, dirty, LRU bits, etc.

- Shared bit: Data is up to date. Other caches may have a copy
- Modified: Data up to date but is dirty. No other cache has copy. Ok to write to Memory
- Exclusive: Data is up to date, no cache has data. Ok to write to memory
- Owner: Up to date data, other caches may have copy but in shared state. Changes need to be broadcasted. Exclusive access to all writes.

8.2.4 OpenMP

OpenMP is a C extension which enables multi-level, shared memory parallelism. It utilizes compiler directives called pragmas to tell the compiler they can parallelize sections of code. OpenMP programs begin as a single process. When a parallel region is encountered, OpenMP forks into parallel threads. These threads are rejoined at the end of the parallel section.

```
#pragma omp parallel for
```

Adding the parallel for pragma is placed above a for loop to parallelize it

```
#pragma omp parallel
```

A regular parallel pragma marks a parallel section. In order to handle locks, OMP uses the critical pragma

```
#pragma omp critical
```

8.2.5 Amdahls Law

While parallization is great, Amdahl's law tells us that there are limits to how fast we can get with parallelization. In particular

$$speedup = \frac{1}{s + \frac{1-s}{p}}$$

where S is the fraction of the program that is serial and P is the factor by which the parallel part is sped up. In the limit, as P grows infinitely large, we see that the maximum speedup we can get is $\frac{1}{s}$, therefore we will always be as slow as the serial part of the program.

9 Input/Output

Input/Output (I/O) is how humans interact with computers. IO devices send data to the CPU via the IO interface. There are two different models of IO.

- Special IO instructions to direct the CPU
- Memory Mapped IO: A portion of the address space is reserved for IO and the CPU uses normal lw, sw operations to access the data

In Memory Mapped IO, we have

- A control register: Says its ok for the CPU to read/write the data register
- Data Register: Contains the data from the IO device

During **Polling**, the processor continuously reads the control register until the data register is ready. After that, it reads the data. However, polling is a waste of processor resources, so for devices which are read from infrequently, we use **Interrupts** and **Exceptions**.

Definition 41 *An interrupt is an external event from the IO device which is asynchronous the current program, meaning it can be handled when the processor finds it convenient*

Definition 42 *An Exception is an external event from the IO device which is synchronous to the current program, requiring it to be interrupted and handled precisely.*

The act of servicing an interrupt or a exception by jumping to new code is called a trap. The trap handler views the state of the program as everything before the last completed instruction, so instructions in the pipeline are considered incomplete. Once the exception is handled, the handler then restores this state by restoring user registers and jumping back to the interrupted instruction.