

# Lecture 1

## Intro, Number Rep

The value of the  $i$ th digit  $d$  in any number base is  $d \times \text{Base}^i$  where  $i$  starts 0 and increases from right to left.

## Signed Magnitude

Similar to unsigned numbers but the most significant bit (first bit) represents if the number is positive or negative, this bit is called the *sign bit*.

The issue with this representation is that there are two representations for the number 0, and operations are slow since there is additional work required to handle the sign bit.

## Two's Complement

The convention to represent signed numbers is called **Two's Complement**. The left most bit is the sign bit, and  $1111 \dots 1111_{\text{two}}$  is the most negative number. The advantage twos complement has over *sign and magnitude* representation is that there is only one zero.

## Negation Shortcut

Simply flip bits, then add 1 to the result. This reason this shortcut works is because the sum of a number and its inverted representation must be  $1111 \dots 1111_{\text{two}}$  (*there is no carries*).

Now since  $x + \bar{x} = -1$ , we have  $\bar{x} + 1 = -x$

## Sign Extension Shortcut

- for positive 16  $\rightarrow$  32 bit binary numbers, just add 16 zeros in the most significant bit
- For negative 16  $\rightarrow$  32 bit binary numbers, copy the sign bit (which is 1) 16 times, placing it on the left of the number.

This works because positive numbers have an infinite number of leading zeros and negative numbers have an infinite number of leading ones.

## One's Complement

A representation in which the negative of a none's complement is found by inverting each bit. So now  $11 \dots 110_{\text{two}}$  is equal to -1. This representation is similar to twos complement but has two 0s

## Biased Notation

A notation that represents the most negative value by  $00 \dots 000_{\text{two}}$  and the most positive value by  $11 \dots 111_{\text{two}}$ , with zero typically having the value  $10 \dots 000_{\text{two}}$ , thereby biasing the number such that the number plus the bias has a nonnegative representation.

In simpler words, bias notation is like unsigned representation but has a shift (a bias), shifting the range of values on the unsigned number line to the left, allowing representation of negative numbers.

With this new system, to interpret a binary string in bias notation, we evaluate the number as if it was unsigned, then add the bias (the bias is usually negative). Note we do this because we shift the numbers to the left by a bias.

### Example 1.1: Interpreting a Stored Binary

Assume we have a  $-127$  bias with an 8-bit number.

To read  $0b0000\ 1001$ , we treat it as if it was unsigned, which gives us 9. Then we add the bias to get our value  $-118$ :

$$9 + (-127) = -118$$

To store a decimal number as a binary string in bias notation, we first subtract the bias (bias is negative so basically add) then store the resulting number as an unsigned binary.

### Lemma 1.2

Subtracting a bias will never give us a negative number.

This is because the range of numbers is from  $[-B, 2^n - 1 - B]$ , where  $B$  is the magnitude of the bias. So a number  $x$ , will always be greater than or equal to  $-B$ , and obviously  $x - (-B) \geq 0$ .

## Questions

- is right shift and left shift division and multiplication by 2 respectively?

## Chapter 0: Introduction

The arguments to functions are passed by copying the value of the argument, and it is impossible for the called function to change the actual argument in the caller. When desired to achieve "call by reference," a pointer may be passed explicitly, and the function may change the object to which the pointer points.

Array name are passed as the location of the array origin, so array arguments are effectively call by reference.

C is not a strongly-typed language. It is relatively permissive about data conversion, although it will not automatically convert data types with the wild abandon of PL/I

## Chapter 2: Types, Operators, and Expressions

A **string constant** is a sequence of zero or more characters surrounded by double quotes. Underneath, a string is an array whos elements are single characters. The compiler automatically places the null character `\0` at the end of each string, so the program can conveniently find the end.

### Bitwise Operations

In C, a leading 0 on an int constant implies *octal*. A leading 0x indicates *hexa-decimal*.

The bitwise AND operator `&` is used to turn off bits.

The bitwise OR operator `|` is used to turn on bits.

The one's operator `~` (one's complement) is used to flip bits.

### 2.1 Memory Model Review

Memory works very similar to an array. You can think of most version of memory you work with conceptually to be one very long array,

#### 2.1.1 The Stack

The stack is memeory that is automatically allocated and freed by the system, and grows from top-down. **What is the stack used for?**

- Anything considered "Temporary"

- This includes: local variables, local constants, arguments to functions, local information about a function call.

### **Stack Frames, function calls**

- Everytime a function is called, a new "stack frame" is allocated on the stack
- Stack Frame Includes:
  - return "instruction" address (who called me?), arguments, and space for other local variables
- When function ends, stack frame is tossed off the stack; automatically frees memory for future stack frames.

### **2.1.2 The Heap**

## **Questions**

- What is an example of local information about a function call?

06/22/23

## **Lecture 3**

*C Pointers, Arrays, Memory Management*

06/27/23

## **Lecture 4**

*C Memory*

Stack grows down, i.e, starts af FF and decreases.

The stack memory is destroyed as you return from functions

The heap memory does not, you are responsible for destorying (freeing) the memory when you don't need it anymore.

## Chapter 5: Pointers and Arrays

### Definition 4.1: Pointer

A pointer is a variable that contains the address of another variable. The convention to naming pointer variables is the prefix them with `p_<name>`.

An pointer that points to an `int` is declared as follows:

```
int *p_x;
```

It says that the combination of `*p_x` is an `int`, that is, if the derefencing operator is used on `p_x`, it is equivalent to a variable of type `int`.

- The unary operator `&` gives the *address* of an object.
- The dereference operator `*` used in the context as an operand to an address, access that address to fetch the contents to which it points to.

### Pointers and Syntax

- pointers can occur on the left side of assignments. That is `p_x` points to `x`, then `*p_x = 1` sets `x` to 1.
- Normally a pointer can point to only one type, however, `void *` is a type that can point to anything. (use sparingly)

### Arrays

Arrays and pointers have a strong relationship. The name of an array is actually an address location to the zeroth element in the collection.

So saying `p_arr = arr` is equivalent to saying `p_arr = &arr[0]`

Moreover, indexing into an array is actually using pointer arithmetic under the hood. For example: doing `arr[3]` is equivalent to `*(a + 3)`.

### Address Arithmetic

The only difference between an array name and a pointer is that a pointer is a variable, but an array name is a **constant**. Constructions like `a = p_arr` or `a++` or `p = &a` are illegal.

### Memory Organization

Memory is just a large, single-dimensional array with the, that is byte-addressable, where the address acting as the index to that array. 8 bits = 1 byte, and 4 bytes = 1 word.

**Type Sizes:** Assume we are working with a 32-bit system.

- `int` occupy 4 bytes, aka 32 bits.
- `char` occupy 1 byte
- *pointers* occupy 4 bytes. This is because memory address are 32 bits long in a 32-bit system (why?), therefore the pointer values, the address that the pointer points to, are also 32 bits long.

#### Definition 4.2: Endianness Systems

Endianness affects how a group of bytes are stored and read in memory.

**Little-Endian:** In a little-endian system, memory is stored with the most significant byte at the highest address.

**Big-Endian:** In a big-endian system, memory is stored with the most significant byte at the lowest address.

#### Example 4.3

For the program below assume, the memory is drawn as such in little endian system.

```
int main() {
    int x[2]; // address of x is: 0x1000 0000
    x[0] = -2, x[1] = 44513;
    char y[] = "ADEL"; // address of y is: 0x1000 0010
    char *c y;
}
```

The corresponding memory model is as follows

Address	Byte 0	Byte 1	Byte 2	Byte 3
0x1000 0000	0xFE	0xFF	0xFF	0xFF
0x1000 0004	0xE1	0xAD	0x00	0x00
0x1000 0008	-	-	-	-
0x1000 000C	-	-	-	-
0x1000 0010	0x41	0x44	0x45	0x4C
0x1000 0014	0x00	-	-	-
0x1000 0018	0x10	0x00	0x00	0x10

**Note:** While ints get stored in reverse, character arrays or strings are stored in increasing memory addresses.

- type declaration tells compiler how many bytes to fetch on each access through pointer.
-

## Dynamic Allocation

- malloc: memory Allocation
- calloc: cleared allocation
- realloc: re-allocation

68 6c 70 74 78 7c 80 84 88 8c 90

## Questions

- is there double pointers, a pointer that points to another pointer? **Yes, you write a function to increment a pointer. Here it will accept a double point as an argument**
- Is there null in C? Can you initialize a pointer to be null? **Yes, C has a null keyword, NULL**
- It is said that the difference between arrays and pointers is that array names is a constant, not a variable. Can you have a pointer to a constant? Why is `p = &arr` illegal?

06/27/23

## Lecture 5

### *Floating Point*

Just as we can show decimal numbers in scientific notation, we can also show binary numbers in scientific notation:

$$1.0_{\text{two}} \times 2^{-1}$$

To keep a number in normalized form, we need a base that allows us to shift the binary point left or right to have on nonzero digit to the left of the decimal point. Only base 2 fulfills this since multiplication by 2 is a left shift and division is a right shift.

### Definition 5.1: Floating point

Computer arithmetic that represents numbers in which the binary point is not fixed, i.e., it floats. The floating-point representation is as such:

31	30	23	22	0
s	exponent			fraction/mantissa

The scientific notation in binary form of the float is a singly nonzero digit to the left of the binary point (**normalized form**) as such:

$$1. \underbrace{xxxxxxxx}_{\text{mantissa/significand}}_{\text{two}} \times 2^{yyyy \rightarrow \text{exponent}}$$

Where  $1.xxxxxxxxx$  is the significand ( $0 < S < 1$ ), and  $yyyy$  is called the exponent.

To pack more numbers in the significand, IEEE 754 makes the leading 1-bit of a normalized binary numbers implicit. So now, the mantissa is 1 + 23-bit significand field. Therefore, the simple RISC-V floating number as such:

$$(-1)^S \times (1 + F) \times 2^E$$

Where  $S$  is the sign of the floating-point number,  $F$  involves the value in the fraction field, generally a number between 0 and 1, also called the *mantissa*. And  $E$  comes from the exponent field.

Note however, float comparisons are hard with this form. The most negative exponent is all 1s, and  $1_{\text{ten}}$  is a single 1 bit in the least significant bit. The negative number looks larger. Therefore we introduce bias notation to shift the numbers where now the most negative number is  $00 \dots 00_{\text{two}}$  and the most positive as  $11 \dots 11_{\text{two}}$ . IEEE 754 uses a bias of 127 for single precision, and 1023 for double precision.

Biased exponent means that the value represented by a floating-point number is really:

$$(-1)^S \times (1 + F) \times 2^{E - \text{Bias}}$$

### Example 5.2

What is the decimal equivalent of the following IEEE 754 single-precision binary floating point number?

31	30	23	22	0
1	1000 0001	111 0000 ... 0000		

$$= (-1)^1 \times 1.111 \times 2^{129-127} \quad (1)$$

$$= -1 \times 1.111 \times 2^2 \quad (2)$$

$$= -1 \times 111.1 \quad \text{shift the decimal point by 2} \quad (3)$$

$$= -1 \times (4 + 2 + 1 + \frac{1}{2}) \quad (4)$$

$$= -7.5 \quad (5)$$



## Step Size

Because we have a fixed # of bits, we cannot represent all numbers. **Step size** is the spacing between consecutive floats with a given exponent.

What we really are asking for is what is the next representable number after  $y$ ? before  $y$ ?

The next step size to why is just adding  $y$  to the smallest bit in the significand times the same exponent.

$$y + ((0.0\dots001) \times 2^{(E-Bias)})$$

Note we multiply by the same exponent because  $y$  decimal was shifted, so we also need to shift the smallest bit decimal to the same position.

*Note:* the bigger the exponent, the bigger the step size since it's shifted more. And the smaller the exponent, the smaller the step size

## Representing Zero

Note: Zero has no normalized representation (there is always an implicit 1 in the significand)

06/28/23

## Lecture 6

*Risc-V Intro*

The operands of arithmetic instructions are restricted, in that they must be from a limited number of special locations built directly in hardware called *registers*.

### Definition 6.1: Registers

Registers are primitives used in hardware design, often thought of as the "bricks of computer construction". The size of a register in RISC-V architecture is 32 bits, aka 1 word.

There is a limited number of registers, 32 registers. The reason for the limit of 32 registers is because of the underlying design principles of hardware; **smaller is faster**. Accessing registers takes less time and uses much less energy than accessing memory.

In code, we usually have more complex data structures such as arrays. These data structures usually contain more data elements than there are registers. How can a computer represent and access such large structures? They are kept in memory and accessed via instructions that transfer data between memory and registers. Such instructions are called **data transfer instructions**.

It is the computer job to associate program variables with registers. Usually a program contains many more variables than computer registers. The compiler tries to keep the most frequently

used variables in registers and places the rest in memory, data transfer instructions to between registers and memory. This process is called *spilling registers*.

06/29/23

## Lecture 7

*RISC-V Procedures*

### Calling Convention

Saved Registers:

- s0-s11 ra
- should not be modified by functions (can be used, but must be restored).

Temporary Registers:

- **TODO**

Stack

07/05/23

## Lecture 8

*RISC-V Instruction Format*

Instructions are kept in the computer as a series of high and low (1s and 0s) electronic signals and may be represented as numbers. Placing these numbers side by side forms the instruction. The numeric version of instructions is called **machine language**, and a sequence of such instructions is called *machine code*.

Instructions in the numeric version are split up into **fields**, which are given names to make them easier to discuss. Here is the meaning of each name of the fields in RISC-V instructions:

- **opcode**: Instruction identifier. The field that denotes the operation and format of an instruction, and is always the last 7 bits in all instruction formats.
- **rd**: The register destination operand. It gets the result of the operation.
- **funct3**: An additional opcode field.
- **rs1**: The first register source operand.
- **rs2**: The second register source operand.
- **funct7**: An additional opcode field.

Different instructions require different fields sizes. For example, add requires 3 registers, addi requires 2 registers and 1 immediate. If rs2 took the value of the immediate, the maximum size it could hold is  $2^5 - 1$  (31 which is pretty small) since rs2 is 5 bits long. Since RISC-V designers decided to keep all instructions the same length (32 bits), this requires distinct instruction formats for different kinds of instructions.

The formats are distinguished by the values in the opcode field (check reference sheet to see the different types of formats).

## R-Type

Designed for instructions with 3 registers and no immediate.

Since each register is identified by its number, we'd need 5 bits to encode 32 different values ( $2^5 = 32$ ).

## I-Type

Designed for instructions with 2 registers (rs1 and rd) and 1 immediate. Some instructions that use I-Types include arithmetic operations with immediates and load instruction.

In the R-Type, there is no funct7 field, instead, that space is used by the immediate to be able to get larger constants (12 bits long immediate)

The immediate field is in unsigned representation, so the range of values is  $[-2048, 2047]$   $[-2^{11}, 2^{11} - 1]$

## I\*-Type

Used for shifting with immediates (slli, srli, srai) where the immediate field is reduced to 5 bits long. This is because in shifting a 32-bit number, you never need to shift more than 32 times.

## S-Type

S for **store instructions**. Designed for instructions with 2 source registers and an immediate.

## U-Type

Used for lui (*load upper immediate*) and auipc (*add upper immediate to program counter*) instructions.

These two instructions are used in two pseudoinstructions:

- li: load immediate
- la: load address

### Example 8.1: li with Large Immediate

consider this instruction:

```
li t0 0x12345678
```

We cannot just translate this instruction to: `addi t0 x0 0x12345678` since the immediate size of `addi` is 5 bits long. Assuming that `lui` has an immediate size of 20 bits (which it does), the solution then is to do this:

```
lui t0 0x12345 // 20 bit long immediate (5 * 4)
addi t0 t0 0x678 // 12 bit immediate (fits in addi immediate field.)
```

*\* Note there is a corner case when the immediate in an `li` instructions has `FFF` in the 3 right most bits, since `addi` with an immediate of `FFF` subtracts  $-1$ .*

## B-Type

B for branch instructions. Recall, labels don't actually exist. When translating RISC-V to machine language, we need to convert all labels into explicit references to a particular line of code. Since we want to be able to move around code blocks in memory, we prefer to use **relative addressing** instead of absolute addresses.

Therefore, when writing code using labels, we first convert the label into an *offset*, which specifies how many bytes off from the current location (PC - program counter) we would need to jump to get to that label.

### Example 8.2: Converting Labels to Offsets

Consider the following code

```
beq x0, x0, target // + 2 instructions = 8 bytes, so offset = 8
addi x0, x0, 100
target: addi x0, x0, 100
j target // - 1 instruction = -4 bytes, so offset = -4
li t0 0x5F3759DF
beq t0, t0, target // -4 instructions = -16 bytes, so offset = -16
```

*\* note `li` is a pseudoinstructions for 2 instructions*

## Storing Offsets:

Note all of the previous offsets were multiples of 4 since each instruction is 4 bytes of memory, so all offsets should be multiples of 4. Therefore, the last two bits off the offset are always going to

be 0s. For optimization purposes, we don't store the lowest bit of an offset immediate, bringing forth the assumption that the LSB of an offset is 0.

In the instruction format, the immediate representing the offset is stored in a strange pattern: `imm[12 | 10:5]` for bits 31 through 25 of the B-Type format, and `imm[4:1 | 11]` for bits 11 through 7.

- if we have the binary 0b A BCDE FGHI JKLM (where each letter was a bit), the first immediate field would store 0b ACD EFGH.
- the second immediate field would store 0b I JKLB. *note bit M isn't stored*

## J-Type

J for jump instruction, `jal`. This instruction uses only 1 destination register and an immediate example: `jal ra label`, so we can use the U-type format to have a larger immediate, but the immediate is stored in a strange format, to simplify the underlying circuit.

The immediate is stored as: `imm[20 | 10:1 | 11 | 19:2]`

For example: If we had the binary 0xA BCDE FGHI JKLM NOPQ RSTU (where each letter was a bit), the data would be stored as 0b AKLM NOPQ RSTJ BCDE FGHI

07/06/23

## Lecture 9

*Compiler, Assembler, Linker, Loader: CALL*

## Compiler

The compiler transforms the C program into an **assembly language** program, a symbolic form of what the machine understands.

### Definition 9.1: Assembly Language

a symbolic language that can be translated into binary machine language.

## Assembler

The assembler converts **pseudoinstructions** into the machine language equivalent of the actual assembly language instruction. It also converts branches to faraway locations into a branch and a jump.

### Definition 9.2: Pseudoinstructions

A common variation of assembly language instructions often treated as if it were an instruction in its own right.

The primary task of an assembler is assembly into machine code. The assembler turns the assembly language program into an *object file*, which is a combination of machine language instructions, data, and information needed to place instructions properly in memory.

To Produce binary version of each instruction into assembly language program, the assembler must determine the address corresponding to all labels. Assemblers keep track of labels used in branches and data transfer instruction in a **symbol table**.

### Linker

The linker produces an **executable file** that can be run on a computer. The linker resolves all references, resolving all undefined labels, patching both internal and external references.

### Loader

A systems program that places an object programs n main memory is that it is ready to execute.

07/10/23

## Lecture 10

*Combinational Logic, FSM*

## A.2: Gates, and Logic Equations

Blocks without memory are called *combinational*; the output of a combinational block depends only on the current input.

In blocks with memory, the output can depend on both the current input and the value stored in memory, which is called the *state* of the logic block.

### Definition 10.1: Combinational Logic

A logic system whose blocks do not contain memory and hence compute the same output given the same input.

Since combinational logic block contains no memory, it can be completely specified by defining the values of the outputs for each possible set of input values. Such a description is normally

given as a *truth table*. Truth tables can completely describe any combinational logic function; however, they grow in size quickly ( $2^n$  entries for  $n$  inputs) making them hard to understand.

## Boolean Algebra

Used to represent signals in logic circuits. One alternative to truth tables is to express the logic function with logic equations. In boolean algebra, all the variables have the values 0 or 1, and in typical formulations, there are three operators:

- OR operator, written as  $A + B$ .
- AND operator, written as  $A \cdot B$ .
- unary NOT operator, written as  $\overline{A}$

Any set of logic functions can be written as a series of equations with an output on the left-hand side of each equation and a formula consisting of variables and the three operators above on the right-hand side.

## Gates

A device that implements basic logic functions, such as AND or OR.

Since both AND and OR are commutative and associative, they can take in multiple inputs, with the output equal to the AND or OR of all the inputs.

Fact: all logic functions can be constructed with only a single gate type, if that gate is inverting. NOR and NAND gates are called *universal*, since any logic function can be built using this one gate type.

## A.3: Combinational Logic

TODO

## FSM: Finite State Machines

TODO

### Questions

- is the memory contained inside the logic block called the state? or is the output called the state?
-

## Lecture 11

### *Synchronous Digital Systems*

Synchronous Digital Systems are made up of two basic types of circuits. Combinational Logic circuits, and state elements. **Combinational logic** circuits change based on their inputs after whatever propagation delay is associated with them. They're like a function.

## State Elements

**State elements**, on the other hand can *remember* their inputs even after the inputs change. State elements change value based on a *clock signal*.

### Clock:

- Rising Edge: Time when the clock switches from 0 to 1
- Falling Edge: Time when the clock switches from 1 to 0.
- Clock Period: Time between rising edges.
- Clock Frequency: Number of rising edges per second.

Like logic gates, registers also have a delay associated with them before their output will reflect the input that was sampled at the rising edge of the clock. This is called **clk-to-q** delay ("Q" often indicates the output.) This is the time between the rising edge of the clock signal and the time the register's output reflects the input change.

## Flip Flop

A Flip Flop is a logic element that 'stores state'. On the rising edge of the clock, the input is sampled and transferred to the output. At all other times, the input is ignored.

For circuits with multiple flip flops, the flip flop responds to the input at a single time instant when the clock is at a rising edge.

### Time Constraints

Flip flop sometimes have what's called time constraints. ( $t_{\text{setup}}$ ,  $t_{\text{hold}}$ ) These are **not** delays, but referred to as constraints because they must be followed in order for the circuit to work properly. Below is the definition of the timing constraints:

#### **setup time:** $t_{\text{setup}}$

- This is the amount of time that the flip flop input needs to be stable **BEFORE** the positive edge of the clock.



**hold time:**  $t_{\text{hold}}$

- the minimum amount of time that the flip-flop input needs to be stable **AFTER** the positive edge of the clock.

### Satisfying setup time constraint (minimum clock period)

Note during a single clock period, there is a clk-to-q time delay, and some logic delay (assuming there is logic elements in the circuit), and finally a setup time constraint for the next rising edge. Therefore, we need to have a clock period to be long enough so that it could hold all of these timing constraints and delays.

$$t_{\text{clk-to-q}} + t_{\text{logic max}} + t_{\text{setup}} \leq t_{\text{clk-period}}$$

### Satisfying hold time constraint

It is easy to see that for a circuit without a logic elements, the hold time for a flipflop should always be less than or equal to the clk-to-q.

However, when circuits contain logic elements, which have delays of their own, we'd need to hold time to be short enough so that it is less than the sum of the clk-to-q and the shortest logic delay path. Therefore, the inequality needed so that we don't violate the hold time constraint is as such:

$$t_{\text{clk-to-q}} + t_{\text{logic min}} \geq t_{\text{hold}}$$

07/12/23

## Lecture 12

*RISC-V Single Cycle Datapath*

For every implementation of instructions; the first two steps are identical, the other steps are dependent on the type of instruction.

1. Send the *program counter* (PC) to the memory that contains the code and fetch the instruction from that memory.
2. Read one or two registers, using fields of the instruction to select the registers to read. For the lw instruction, we need to read only one register, but most other instructions require reading two registers.
3. execute the operation or calculate an address
4. Access an operand in data memory (if necessary)

5. Write the result into a register (if necessary)

After these two steps, the actions required to complete the instruction depend on the instruction class (memory-reference, arithmetic-logical, or branches).

### Definition 12.1: Control Unit

A *control unit* has an instruction as an input, and is used to determine how to set the control lines for the functional units and two of the multiplexors.

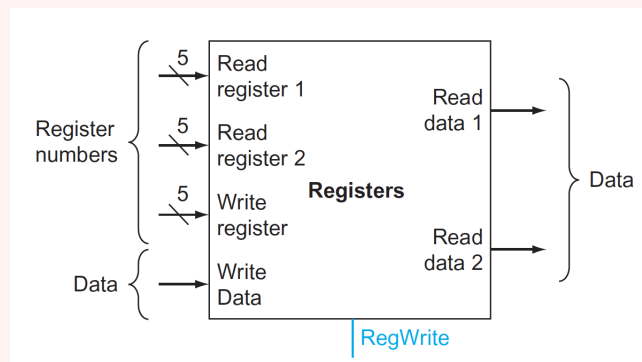
## Building a Datapath

To execute any instruction, we must start by fetching the instruction from memory (code section of the memory?) To prepare for executing the next instruction, we must also increment the PC so that it points at the next instruction, 4 bytes later.

The processor's 32 general-purpose registers are stored in a structure called a **register file**.

### Definition 12.2: Register File

A state element that consists of a set of registers that can be read and written by supplying a register number to be accessed. A register file always outputs the contents of the registers corresponding to the Read register inputs on the outputs; no other control inputs are needed. However, a register write must be explicitly indicated by asserting the write control signal. This means that a register file is a combinational read, but a sequential write.



Note: the 5 on top of the register number arrows indicate that the register numbers is 5 bits wide, and those bits come from a state element called *instruction memory*, which is where the machine code of our instructions lie.

## R-Type Instructions

Recall R-format instructions read two registers (*rs1* and *rs2*), perform an ALU operation on the contents of the registers and write the result to a register (*rd*): `add rd rs1 rs2`

With that, this means that we need to *read* two data words from the register file and *write* one data word into the register file for each R type instruction.

### Reading:

For each word to be read from the registers, we need an input to the register file that specifies the *register number* to be read and an output from the register file that will carry the value that has been read from the registers (**two outputs? yes.**)

### Writing:

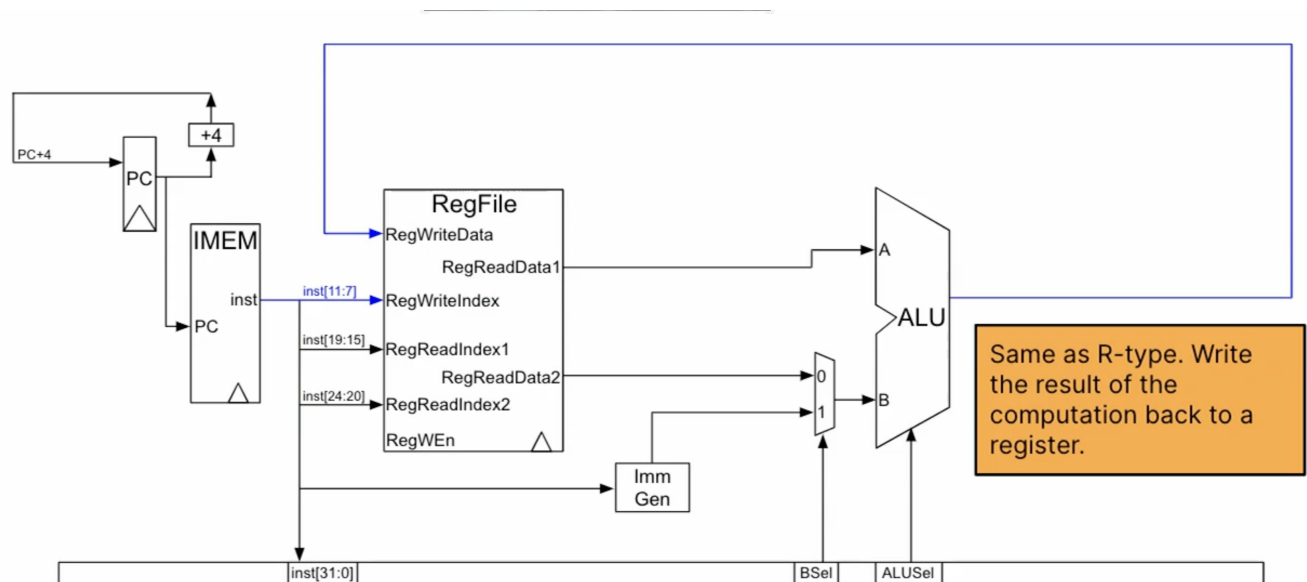
To write a data word, we will need two inputs: one to specify the register number to be written and one to supply the *data* to be written into that register. Writes are controlled by the write control signal, which must be **asserted** for a write to occur at the clock edge.

## I Type Instructions

addi

Note that for an addi, the output **Read Data 2** from the register file will contain a garbage value since **rs2** is not a field of I type instructions, instead it has a 12 bit immediate.

We therefore directly extract the immediate from the instruction memory, and put an arrow directly to the ALU. Moreover, we'd need a unit to **sign extend** the immediate, call this unit *immediate generator* and a multiplexor to select what will go into the ALU, the Read data 2, or the immediate.



**Figure 12.1:** Data path for addi instruction.

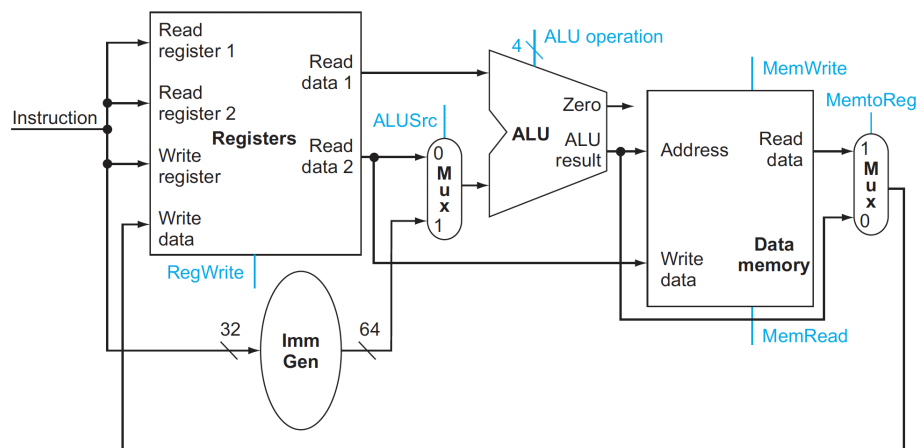
## Load and Store instructions

Note a load instruction is a I type instruction with the structure as such:

```
lw rd imm(rs1)
```

where  $rs1 + imm$  is the address we are reading from and we write the value at that address to rd. And a store instruction is similar except instead of writing to a register, we are writing to memory.

The current data path does not work for load and store instructions, why? Because we don't have a unit for **data memory** to read from and write to. So let's introduce it.



**Figure 12.2:** The datapath for memory instructions and the R-type instruction

07/17/23

## Lecture 13

*Single Cycle Datapath Controls, Intro to Pipelining*

## Pipelining

### Definition 13.1: Pipelining

Pipelining is one example of instruction-level parallelism, an implementation technique in which multiple instructions are overlapped in execution.

Note pipelining does not decrease the time to complete a single instruction, it instead increases instruction *throughput* since it allows multiple instructions to be processed concurrently, decreasing the total time to complete the work.

## Pipeline Hazards

There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called *hazards*, and there are three different types.

...

**Lecture 14**  
...

07/20/23

**Lecture 15**  
*Parallelism I - Data-Level*

## Terminology

- Processors are often called cores in a multicore chip.
- SIMD: Single Instruction, Multiple Data. SIMD computers operate on vectors of data. These vectors are specialized vector registers which store 128, 256, or even 512 bits.
- For example, a SIMD instruction might add 64 numbers by sending 64 data streams to 64 ALUs to form 64 sums within a single clock cycle.

## Chapter 6.1: Introduction

- Computer architects seek to create powerful computers by connecting many smaller ones, leading to the concept of multiprocessors.
- Multiprocessors can also improve system availability by supporting operation even in the presence of hardware failures.
- Multicore microprocessors, which are expected to increase with Moore's Law, are mostly Shared Memory Processors (SMPs) sharing a single address space.
- The industry faces the challenge of creating hardware and software that can efficiently execute parallel processing programs with an increasing number of cores.

## Chapter 6.2: The Difficulty of Creating Parallel Processing Programs

- Creating efficient parallel processing programs has proven to be challenging, particularly as the number of processors increases.
- Uniprocessor design techniques, such as instruction-level parallelism, have reduced the need for rewriting programs for multiprocessors.

- Efficient parallel programs require breaking tasks into equal-sized pieces, load balancing, and addressing synchronization and communication overhead.
- Amdahl's Law reminds us that even small portions of a program must be parallelized to fully utilize many cores.

### **Chapter 6.3: SISD, MIMD, SIMD, SPMD, and Vector**

- Parallel hardware can be categorized based on the number of instruction streams and data streams.
- SISD (Single Instruction Single Data) refers to conventional uniprocessors with one instruction stream and one data stream.
- MIMD (Multiple Instruction Multiple Data) refers to conventional multiprocessors with multiple instruction streams and multiple data streams.
- SPMD (Single Program Multiple Data) is a programming style for MIMD computers where a single program runs on all processors.
- SIMD (Single Instruction Multiple Data) operates on vectors of data, executing the same instruction simultaneously on multiple data streams.
- SIMD's virtues include synchronized execution units and reduced instruction bandwidth.
- SIMD works best for data-level parallelism, where arrays are processed in for loops.
- SIMD's weakness lies in case or switch statements, leading to reduced performance.
- SIMD in x86 architecture includes Multimedia Extensions (MMX), Streaming SIMD Extensions (SSE), and Advanced Vector Extensions (AVX).
- Vector architectures, a form of SIMD, involve pipelined execution units and a set of vector registers.
- Vector instructions specify a significant amount of work, reducing dynamic instruction bandwidth and control hazards.
- Vector architectures are known for their ease in handling data-level parallelism.
- Vector instructions accessing memory have a known access pattern, benefiting memory access efficiency.
- Vector units are advantageous if the application domain can often use them.
- Vector architectures and multimedia extensions differ in the number of operations specified and the handling of data transfers.
- Vectors support both strided and indexed accesses for data in memory.

- Vector arithmetic instructions usually operate on specific elements of vector registers, facilitating parallelism through multiple vector lanes.

07/20/23

## Lecture 16

*Parallelism II - Thread-level*

### Definition 16.1: Hardware Multithreading

Increasing utilization of a processor by switching to another thread when one thread is stalled. Hardware multithreading allows multiple threads to share the functional units of a *single* processor in an overlapping fashion to try to utilize the hardware resources efficiently. Note to permit this sharing, the processor must duplicate the independent state of each thread.

### Definition 16.2: Thread

A thread includes the program counter, the register state, and the stack. It is a lightweight

### SIMD: Single Instruction, Multiple Data

- Computer applies a single stream to multiple data streams.
- Note this will require multiple processing units (1 for each element in the vector) in order to compute the result of a vector operation.

### MISD: Multiple Instruction, Single Data

- Computer applies multiple different operations to the same data.

### MIMD: Multiple Instruction, Multiple Data

- Computer applies multiple different operations to multiple different data.

# Parallel Threads

## Definition 16.3: Threads

A thread stands for "thread of execution", is a single stream of instructions.

- a program / process can split, or fork itself into separate threads, which can (in theory) execute simultaneously.

With a single core, a single CPU can execute many threads by **time sharing**. (note a single core can run one thread at any given time.)

The CPU is composed of multiple cores, thus allowing multiple threads to be run simultaneously.

Each core provides one or more hardware threads that actively execute instructions

The **Operating System** is responsible for managing which threads get run on which CPUs (among other tasks)

## Parallel Programming

. In order to create a parallel section: you use `#pragma omp parallel ...`

The code in the parallel section gets run on all threads, so we need some way to distinguish threads

Recall that OS chooses which ever thread it wants to run, and can change threads at any time. Therefore a program that uses multithreading is no longer deterministic, i.e. it has random behavior.

A multithreaded program is only considered correct if ANY interlacing of threads yield the same result.

In general with OpenMP, variables declared inside the parallel block will be private to each thread, but variables declared outside a parallel block will be shared across all threads

OpenMP has built-in functionality for dividing up the work of for loops among threads. To tell openMP to split up the work among multiple threads, you would use `#pragma omp parallel for`



## Lecture 17

...

07/25/23

## Lecture 18

*Caches I*

Caches – a safe place to store things. *Cache* was the name chosen to represent the level of the memory hierarchy between the processor and main memory.

The simplest way to assign a location in the cache for each word in memory is to assign the cache location based on the address of the word in memory. This cache structure is called **direct mapped**, since each memory location is mapped directly to exactly one location in the cache. Almost all direct-mapped caches use this mapping to the a block:

$$(\text{Block address}) \bmod (\text{Number of blocks in the cache})$$

If the cache size (number of blocks in the cache) is a power of 2, then modulo can be computed simply by using the low-order  $\log_2$  (cache size in blocks) bits of the address. For example, an 8 block cache uses the 3 lowest bits ( $2^3 = 8$ ) of the block address to index the blocks.

Note that because multiple memory locations can map to the same cache location, we need a way to know whether the data in the cache corresponds to a requested block/word. We solve this by adding a set of **tags**.

### Definition 18.1: Tag

A field in a table used for a memory hierarchy that contains the address information required to identify whether the associated block in the hierarchy corresponds to a requested word. Tags therefore will need just to contain the upper portion of the address, i.e., the bits not used as an index into the cache.

We also need a way to recognize that a cache block does not have valid information. We can't just set the blocks to all zeros at indices where the cache is not valid since all zeros might actually be a valid entry. Thus, the most common method is to add a separate **valid bit**.

### Definition 18.2: Valid Bit

A field in the tables of a memory hierarchy that indicates that the associated block in the hierarchy contains valid data.

## How is the hierarchy managed?

1. Communication between Registers ↔ Memory: By **compiler** (or assembly level programmer)
2. Communication between Cache ↔ Main memory: By the **cache controller hardware**.
3. Main Memory ↔ Disks: By the **OS** (virtual memory)

## Locality

Main idea: predict what memory might be accessed next based on what memory was already accessed.

The two types of locality are:

1. Temporal Locality
  - Means: if a memory location is referenced then it will tend to be referenced again soon.
  - Consequences: keep most recently accessed data items in our cache
2. Spatial Locality
  - Means: if a memory location is referenced, the locations with nearby addresses will tend to be referenced soon.
  - Consequences: move blocks consisting of contiguous words instead of just one word at a time.
  - To facilitate this, we will divide memory into "blocks" of size some power of 2.
  - Blocks will be assigned a number, called their tags

## Fully Associative Cache

When a memory access occurs, here are the steps that happen.

1. Check if the cache contains the data needed. If so, return the data.
2. If the data is not present in the cache, load the block of memory into the cache.
3. Return the data.

A fully associative cache is parameterized on two aspects: the block size, and the number of blocks that can be stored in the cache.

### Example 18.3

**Lets say we have a cache of 4 byte blocks and 4 blocks of storage. What is the size of the cache?**

*since the cache stores 4 blocks, and each block is 4 bytes. The size of the cache is 16 bytes*

**How do we access address 0x3C1**

*we split the address into a tag and offset. Since each block is 4 bytes long, the offset needs to be 2 bits inorder to index into those bytes of data ( $2^2 = 4$ ).*

$$0x3C1 = 0b \overbrace{1111\ 0000}^{\text{tag}} \underbrace{01}_{\text{offset}}$$

*after which we scan the tags in the table for matches, and if the valid bit is asserted, then our data is in the cache, so we get the byte at the offset (0b01 = 1)*

**What if the valid bit was off?**

*This is called a **cache miss**. We need to go into main memory and load the block into the cache*

**What if there was not tag matches and the cache is already full of valid data?**

*This is called a **cache miss with eviction**. There is different eviction policies used for which block we replace, such as LRU, FIFO, or Random*

07/26/23

## Lecture 19

*Caches II*