University of Waterloo

CS241 - Winter 2024 - Course Notes

Author: Brandon Zhou

Course Code: CS241

Course Name: Foundations of Sequential Programs

Instructor: Chengnian Sun

Section: 001

Date Created: January 09, 2024

Final Exam Date: TBA

Disclamer: These course notes are intended to supplement primary instructional materials and facilitate learning. It's worth mentioning that some sections of these notes might have been influenced by ChatGPT, an OpenAI product. Segments sourced or influenced by ChatGPT, where present, will be clearly indicated for reference.

While I have made diligent efforts to ensure the accuracy of the content, there is a potential for errors, outdated information, or inaccuracies, especially in sections sourced from ChatGPT. I make no warranties regarding the completeness, reliability or accuracy of the notes contained in this notebook. It's crucial to view these notes as a supplementary reference and not a primary source.

Should any uncertainties or ambiguities arise from the material, I strongly advise consulting with your course instructors or the relevant course staff for comprehensive understanding. I apologize for any potential discrepancies or oversights.

Any alterations or modifications made to this notebook after its initial creation are neither endorsed nor recognized by me. For any doubts, always cross-reference with trusted academic resources.

Table of Contents

Section 1: Lectures

Lecture 1

- Definition: A bit is a binary digit. That is, a 0 or 1 (off or on)
- Definition: A nibble is 4 bits.
  - Example: 1001
- Definition: A byte is 8 bits.
  - Example: 10011101
- in C/C++:
  - Char: 8 bits
  - Unsigned char: 8 bits
  - Short: 2 bytes/16 bits
  - int: 4 bytes
  - longlong: 16 bytes
- Definition: A word is a machine-specific grouping of bytes. For us, a word will be 4 bytes (32-bit architecture) though 8-byte (or 64-bit architectures) words are more common now.
- Definition (Hexadecimal Notation): The base-16 representation system is called the hexadecimal system. It consists of the numbers from 0 to 9 and the letters a, b, c, d, e, f (which convert to the numbers from 10 to 15 in decimal notation)
  - Sometimes we denote the base with a subscript like 10011101_2 and $9d_{16}$.
  - Also, for hexadecimal,you will routinely see the notation `0x9d`. (The `0x` denotes a hexadecimal representation in computer science).
  - Note that each hexadecimal character is a nibble (4 bits).
- Conversion Table
  - Note: upper case letters are also used for hexadecimal notation. Context should make things clear.

Binary	Decimal	Hex
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7

Binary	Decimal	Hex
1000	8	8
1001	9	9
1010	10	a
1011	11	b
1100	12	c
1101	13	d
1110	14	e
1111	15	f

- Notation:
  - Binary: 0, 1
  - Decimal(Base 10): 0, ..., 9
  - Hexadecimal: 0, ..., 9, A(10), B(11), C(12), D(13), E(14), F(15)
- Examples:
  - 0000(base 2) -> 0x0(base 16)
  - 1111(base 2) -> 0xf(base 16)
- What do bytes represent?
  - Numbers
  - Characters
  - Garbage in memory
  - Instructions (Words, or 4 bytes, will correspond to a compute instruction in our computer system)
- Bytes as Binary Numbers
  - Unsigned (non-negative integers)
    - $b_7 ... b_0$ (base 2) = $b_7 \times 2^7 + \ldots + b_0 \times 2^0$ (base 10)

- Example: $01010101 = 0 \times 2^7 + ... + 1 \times 2^0$
- Converting to Binary:
  - One way: Take the largest power of 2 less than the unsigned integer, subtract and repeat
  - Another way is to constantly divide by 2, get the remainer for each division, reading *from the bottom to up* at the end, and that will be the binary representation of this unsigned integer
    - Example: 38

Number	Quotient	Remainder
38	19	0
19	9	1
9	4	1
4	2	0
2	1	0
1	0	1

    - Brief Explanation: Consider

$$N = b_0 + 2b_1 + 2^2 b_2 + \dots$$

The remainder when dividing N by 2 gives the b_0 value. After doing $\frac{N-b_0}{2}$, we end of with

$$\frac{N - b_0}{2} = b_1 + 2b_2 + 2^2 b_3 + \dots$$

and we can repeat the process. (This is why we have to read bottom-up as we get b_0 first, then b_1...)

- Signed integers
  - Attempt 1: make the first bit a signed bit. This is called the "sign-magnitude" representation
    - Problems:
      - Two representations of 0(wasteful and awkward)
      - Arithmetic is tricky. Is the sum of a positive number and a negative number positive or negative? It depends!
  - Attempt 2: Two's complement form
    - Similar to "sign-magnitude" representation in spirit, first bit is 0 if non-negative, 1 if negative
    - Negate a value by just subtracting from zero and *letting it overflow*.
    - Decimal to Two's Compliment:
      - A trick to the same thing of negating a value:
        - Take the complement of all bits (flip the 0 bits to 1 and 1 bits to 0)
        - Add 1
      - A slightly faster trick is to locate the rightmost 1 bit and flip all the bits to the left of it
        - Example: 11011010 Negating: 00100110 = 00100101 + 1
        - Note: Flipping the bits and adding 1 is the same as
          - subtracting 1 and flipping the bits for non-zero numbers
          - subtracting from 0
        - Example: compute -38_{10} using this notation in one byte of space:
          - Step 1: $38_{10} = 00100110_2$
          - Step 2: take the complment of all the bits: 11011001_2
          - Step 3: plus 1: 11011010_2
    - Two's Compliment to Decimal
      - Let's compute -38_{10} using one-byte Two's complement. First, write 38 in binary: 38_{10} = 00100110_2. Next, take the complement of all the bits 11011001_2. Finally, add 1: 11011010_2. This last value is -38_{10}.
      - To convert 11011010_2, a number in Two's complement representation, to decimal, one method is to flip the bits and add 1: 00100110_2 = $2^5 + 2^2 + 2^1 = 38$. Thus, the corresponding positive number is 38 and so the original number is -38.

        - Another way to do this computation is to treat the original number 11011010_2 as an unsigned number, convert to decimal and subtract 2^8 from it (since we have 8 bits, and the first bit is a 1 meaning it should be a negative value). This also gives -38

$$\begin{aligned}11011010_2 &= 2^7+2^6+2^4+2^3+2^1-2^8\\&=128+64+16+8+2-256\\&=218-256\\&=-38\end{aligned}$$

        - The idea behind [one byte] Two's Complement notation is based on the following observations:
            - The range for unsigned integers is 0 to 255. Recall that 255 is 11111111_2. If we add 1 to 255, then, after discarding overflow bits, we get the number 0.
            - Thus, let's treat 2^8 as 0, i.e., let's work modulo $2^8 = 256$. In this vein, we set up a correspondence between the positive integer k and the unsigned integer 2^8-k. Since we are working modulo 2^8, subtracting a positive integer k from 0 is the same as subtracting it from 2^8.
            - In this cse, note that $255 = 2^8-1 = 2^7+2^6+2^5+2^4+2^3+2^2+2^1+2^0$ and in general

$$2^n-1=\sum_{i=0}^{n-1}2^i$$

            - As an explicit example (which can be generalized naturally) take a number, say $38_{10} = 00100110_2 = 2^5+2^2+2^1$. What should the corresponding negative number be? Well, note that we've said subtracting a positive integer k from 0 is the same as subtracting it from 2^8:

$$\begin{aligned}2^8-1&=2^7+2^6+2^5+2^4+2^3+2^2+2^1+2^0\\2^8-1&=38+2^7+2^6+2^4+2^3+2^0\\2^8-38&=2^7+2^6+2^4+2^3+2^0+1 \qquad \text{(flip the bits and add 1)}\end{aligned}$$

            - We mentioned that another method of negating a two's complement number is to flip the bits to the left of the rightmost 1. Justify why this technique works.
                - Every bit to the right of the rightmost 1 is a 0. When we "flip the bits", these become 1s. When we "add 1", the carry propagates up until the position of the "rightmost 1" (the "rightmost 1" is 0 after flip and will stop propagating when the carry reaches this point, and everything on the left of the "rightmost 1" is flipped).

    - The main difference between signed and unsigned binary arithmetic is that we are now working modulo 256 (or, more generally, 2^n in the case of n-bit Two's complement numbers)
    - When working in Two's complement, overflow occurs when adding numbers if the original two numbers have the same sign, but the result has a different sign.
    - Arithmetic of Signed Integers
        - All of the arithmetic works by ignoring overflow precisely because arithmetic works in Z_{256}!
    - What is the range of numbers expressible in one-byte Two's Complement notation?
        - $-128 \sim 127$
    - Definitions: The Most Significant Bit (MSB) is the left- most bit (highest value/sign bit); The Least Significant Bit (LSB) is the right-most bit (lowest value).

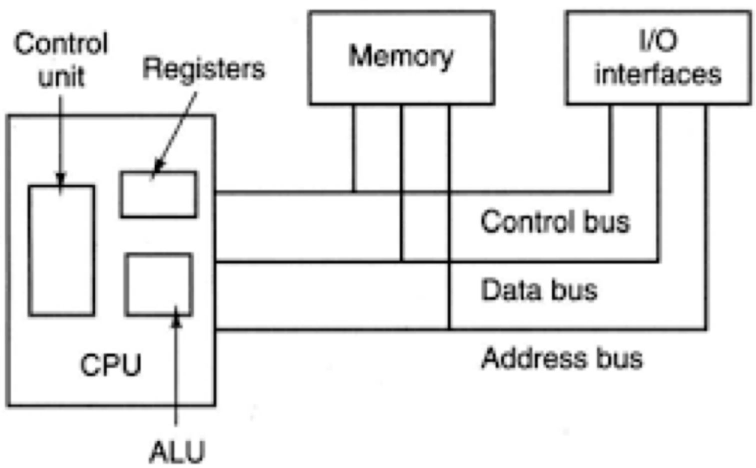
Lecture 2

- ASCII (American Standard Code for Information Interchange) uses 7 bits to represent characters.

```
char c = '0';
printf("%c", c);
```

```
// stdout: 0
printf("%d", (int)c);
// stdout: 48
```

- Bit-Wise Operators
  - Example: suppose we have `unsigned char a=5, b=3;`, which means $a = 00000101, b = 00000011$
    - Bitwise not $\sim a$, for example $c = \sim a$; gives $c = 11111010$
    - Bitwise and `&`, for example `c=a&b;` gives $c = 00000001$
    - Bitwise or $|$, for example $c = a|b$; gives $c = 00000111$
    - Bitwise exclusiveˆ(only true if the bit in a and b is different), for example $c = a\hat{}b$; gives $c = 00000110$
    - Bitwise shift right or left $>>$ and $<<$, for example
      - $c = a >> 2$; gives $c = 00000001$ and
      - $c = a << 3$; gives $c = 00101000$
    - $a << 1$ equivalent to $a * 2$
    - $a >> 1$ equivalent to $a/2$
    - These can even be combined with the assignment operator: `c &= 5;`



- CPU with Memory
- MIPS has 32 registers that are called "general purpose"
  - Some general-purpose registers are special:
    - `$0` is always 0
    - `$31` is for return address
    - `$30` is our stack pointer
    - `$29` is our frame pointer
- Problem: We only know from context what bits have what meaning, and in particular, which are instructions.
  - Solution: Convention is to set memory address 0 in RAM to be an instruction. `0x0: instruction i1`
- Problem: How does MIPS know what to do next?
  - Solution: Have a special register called the Program Counter (or PC for short) to tell us what instruction to do next.
- Problem: How do we put our program into RAM?
  - Solution: A program called a loader puts our program into memory and sets the PC to be the first address.
- Algorithm 1 Fetch-Execute Cycle

```
PC=0
while true do
    IR = MEM[PC]
    PC += 4
    Decode and execute instruction in IR
end while
```

- Write a program in MIPS that adds the values of registers `$8` and `$9` and stores the result in register `$3`.

```
add $d, $s, $t
0000 00ss ssst tttt dddd d000 0010 0000
```

- Why 5 bits for each? Because there are 32 registers `$0 - $31` and $2^5 = 32$
- Adds registers `$s` and `$t` and stores the sum in register `$d`. Important! The order of `$d`, `$s` and `$t` are shifted in the encoding.

Lecture 3

- **Putting values in registers** Load immediate and skip. This places the next value in RAM [an immediate] into $d and increments the program counter by 4 (it skips the next line which is usually not an instruction).

```
lis $d: 0000 0000 0000 0000 dddd d000 0001 0100
# What it really does is
$d = MEM[PC]
PC = PC + 4
```

- How do we get the value we care about into the next location in RAM?

```
.word i: iiii iiii iiii iiii iiii iiii iiii iiii
```

- The above is an assembler directive (not a MIPS instruction). The value i, as a two's complement integer, is placed in the correct memory location in RAM as it occurs in the code.
  - Can also use hexadecimal values: `0xi`
  - Decimal is also allowed.
  - `.word` is not a true opcode, as it does not necessarily encode a MIPS instruction at all. It is a directive for the assembler, telling the assembler to encode a 32-bit word at the location of the directive.
- Example:

```
lis $1
.word 10
# At this moment, $1 = 10
add $1, $1, $0
```

- Example: Write a MIPS program that adds together 11 and 13 and stores the result in register $3.

```
lis $1
.word 11
list $2
.word 13
add $3, $1, $2
# Solution on the course notes
lis $8          0000 0000 0000 0000 0100 0000 0001 0100
.word 11        0000 0000 0000 0000 0000 0000 0000 1011
lis $9          0000 0000 0000 0000 0100 1000 0001 0100
.word 0xd       0000 0000 0000 0000 0000 0000 0000 1101
add $3,$8,$9    0000 0001 0000 1001 0001 1000 0010 0000
    # The code on the left is what we call Assembly Code.
    # The code on the right is what we call Machine Code.
```

- Jump Register. Sets the PC to be $s.

```
jr $s
0000 00ss sss0 0000 0000 0000 0000 1000
```

- For us, our return address will typically be in $31, so we will typically call the below. This command returns control to the loader.

```
jr $31
0000 0011 1110 0000 0000 0000 0000 1000
```

- So the complete example for the example is (so that the while loop will terminate)

```
lis $8          0000 0000 0000 0000 0100 0000 0001 0100
.word 11        0000 0000 0000 0000 0000 0000 0000 1011
lis $9          0000 0000 0000 0000 0100 1000 0001 0100
.word 0xd       0000 0000 0000 0000 0000 0000 0000 1101
add $3,$8,$9    0000 0001 0000 1001 0001 1000 0010 0000
jr $31          0000 0011 1110 0000 0000 0000 0000 1000
```

- To multiply two words, we need to use the two special registers `hi` and `lo`.
  - `hi` is most significant 4 bytes
  - `lo` is least significant 4 bytes

```
mult $s, $t
0000 00ss ssst tttt 0000 0000 0001 1000
```

- The above performs the multiplication and places the most significant word (largest 4 bytes) in `hi` and the least significant word in `lo`.
- `div $s, $t` performs integer division and places the quotient s/t in `lo` [lo quo] and the remainder st in `hi`. Note the sign of the remainder matches the sign of the divisor stored in $s.

```
div $s, $t
0000 00ss ssst tttt 0000 0000 0001 1010
```

- Multiplication and division happen on these special registers hi and lo. How can I access the data?
  - Move from register `hi` into register `$d`.

```
mfhi $d
0000 0000 0000 0000 dddd d000 0001 0000
```

  - Move from register `lo` into register `$d`.

```
mflo $d
0000 0000 0000 0000 dddd d000 0001 0010
```

- RAM
  - Large[r] amount of memory stored off the CPU.
  - RAM access is slower than register access (but is larger, as a tradeoff).
  - Data travels between RAM and the CPU via the bus.
  - Modern day RAM consists of in the neighbourhood of 10^{10} bytes.
  - Instructions occur in RAM starting with address 0 and increase by the word size (in our case 4).
    - But, this simplification will vanish later...
  - Each memory block in RAM has an address; say from 0 to $n-1$
  - Words occur every 4 bytes, starting with byte 0. Indexed by $0, 4, 8, ... n-4$.
  - Words are formed from consecutive, aligned (usually) bytes.
  - Cannot directly use the data in the RAM. Must transfer first to registers.
- Load word. Takes a word from RAM and places it into a register. Specifically, load the word in `MEM[$s + i]` and store in `$t`.

```
lw $t, i($s)
1000 11ss ssst tttt iiii iiii iiii iiii
# which is equivalent to
$t = MEM[$s + i]
# Example
lw $1, -4($30)
# which means
$1 <- MEM[$30 - 4]
```

- Store word. Takes a word from a register and stores it into RAM. Specifically, load the word in `$t` and store it in `MEM[$s + i]`.

```
sw $t, i($s)
1010 11ss ssst tttt iiii iiii iiii iiii
```

- Note that `i` must be an immediate, NOT another register! It is a 16-bit Two's complement immediate
- Example: Suppose that `$1` contains the address of an array of words, and `$2` takes the number of elements in this array (assume less than 220). Place the number 7 in the last possible spot in the array.

```
# First element in the array is arr, then the second element is arr + 4 ... the last element in the array is arr +
4 * (length - 1)
lis $8 ; 7
.word 7
lis $9 ; 4
.word 4
mult $2, $9 ; length * 4
mflo $3 ; length * 4
add $3, $3, $1 ; arr + length 4
sw $8, -4($3) ; MEM[$3 - 4] = $8
jr $31
```

- Branch on equal. If `$s==$t` then `pc += i*4`. That is, skip ahead i many instructions if `$s` and `$t` are equal.

```
beq $s, $t, i
0001 00ss ssst tttt iiii iiii iiii iiii
```

```
# It is like
if ($s == $t) {
    PC += i*4
}
```

- Branch on not equal. If `$s!=$t` then `pc+=i*4`. That is, skip ahead i many instructions if `$s` and `$t` are not equal.

```
bne $s, $t, i
0001 01ss ssst tttt iiii iiii iiii iiii
# It is like
if ($t != $s) {
    PC += i*4
}
```

- Example:

```
beq $0, $0, 0 ; This executes the next instruction as PC has been updated to +4 already
beq $0, $0, 1 ; This executes the second next instruction
```

Lecture 4

- Write an assembly language MIPS program that places the value 3 in register `$2` if the signed number in register `$1` is odd and places the value 11 in register `$2` if the number is even.

```
lis $8 ; $8 = 2
.word 2
lis $9 ; $9 = 3
.word 3
lis $2 ; $2 = 11
.word 11
div $1, $8
mfhi $3
beq $3, $0, 1
add $2, $9, $0
jr $31
```

- Set Less Than. Sets the value of register `$d` to be 1 provided the value in register `$s` is less than the value in register `$t` and sets it to be 0 otherwise.

```
slt $d, $s, $t
0000 00ss ssst tttt dddd d000 0010 1010
# which basically means
if ($s < $t) {
    $d = 1 (true)
} else {
    $d = 0 (false)
}
```

- Example: Write an assembly language MIPS program that negates the value in register `$1` provided it is positive.

```
slt $2, $1, $0
bne $2, $0, 1
sub $1, $0, $1
jr $31
```

- Exercise: Write an assembly language MIPS program that places the absolute value of register $1 in register 2$.

```
add $2, $1, $0 ; $2 = $1
slt $3, $0, $1 ; 0 < $1
bne $3, $0, 1
sub $2, $0, $2 ; $2 = 0 - $2
jr $31
```

- Looping example: Write an assembly language MIPS program that adds together all even numbers from 1 to 20 inclusive. Store the answer in register `$3`.
  - Note: semicolons for comments in MIPS assembly

```
lis $2
.word 20
lis $1
.word 2
add $3, $0, $0
add $3, $3, $2 ; line -3
sub $2, $2, $1 ; line -2
bne $2, $0, -3 ; line -1 from here
jr $31
```

- Labels aren't machine code, so don't take words. That means that for `beq` and `bne`, *labels don't have "line numbers" on their own*. A label at the end of code is allowed. It has the address of what would be the first instruction after the program.

```
label: operation commands
```

- Example: `sample` has the address $0x4$, which is the location of `add $1, $0, $0`.

```
sub $3, $0, $0
sample:
add $1, $0, $0
```

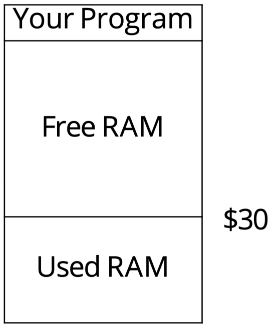
- A better way to loop without hard-coding -3 in the previous example is (otherwise, if we were to, say, add a new instruction in between the lines specified by our branching, all our numbers would be incorrect.)
    - Note that `top` in `bne` is computed by the assembler to be the *difference between the program counter and top*. That is, here it computes $(top - PC)/4$ which is $(0\text{x}14 - 0\text{x}20)/4 = -3$
    - PC is the line number after the current line

```
lis $2
.word 20
lis $1
.word 2
add $3, $0, $0
top:
    add $3, $3, $2
    sub $2, $2, $1
    bne $2, $0, top
jr $31
```

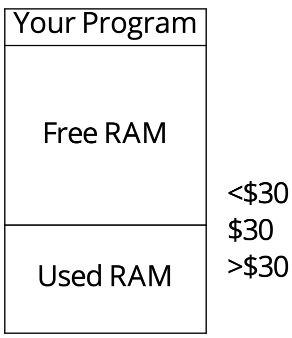
- RAM

- Register `$30` initially points to the very bottom of the free RAM. It can be used as a bookmark to separate the used and unused free RAM if we allocate from the one end, and push and pop things like a stack! In other words, we will use `$30` as a pointer to the top of a stack.

- Really, `$30` points to the top of the stack of memory in RAM.

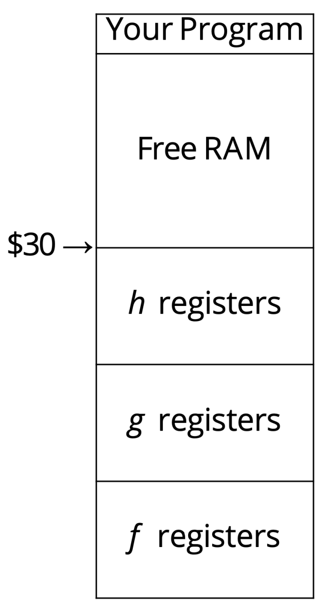


- Because our program is at zero, the stack grows from high memory to low memory, so pushing involves reducing the value of `$30`.



- Example: Suppose procedures f, g and h are such that:

```
f calls procedure g
    g calls procedure h
        h returns
    g returns
f returns.
```



- In the previous example:
  - Calling procedures pushes more registers onto the stack and returning pops them off.
  - This is a stack, and we call `$30` our **stack pointer**.
  - We can also use the stack for local storage if needed in procedures. Just reset `$30` before procedures return.
- Template for a procedure `f` that modifies registers `$1`and `$2`:

```
f:
sw $1, -4($30) ; Push registers we modify
sw $2, -8($30)
lis $2 ; Decrement stack pointer
.word 8
sub $30, $30, $2
    ; Insert procedure here
add $30, $30, $2 ; Assuming $2 is still 8
lw $2, -8($30) ; Pop = restore
lw $1, -4($30)
    ; Uh oh! How do we return?
```

- There is a problem with returning:

```
main:
lis $8
.word f ; Recall f is an address
jr $8 ; Jump to the first line of f
(NEXT LINE)
```

- Once f completes, we really want to jump back to the line labelled above as (NEXT LINE), i.e., set the program counter back to that line. How do we do that?
- Jump and Link Register. Sets `$31` to be the PC and then sets the PC to be `$s`. Accomplished by `temp = $s` then `$31 = PC` then `PC = temp`.

```
jalr $s
0000 00ss sss0 0000 0000 0000 0000 1001
```

- Main Changes

```
main:
    lis $8
    .word f
    sw $31, -4($30) ; Push $31 to stack
    lis $31 ; Use $31 since it's been saved
    .word 4
    sub $30, $30, $31
    jalr $8 ; Overwrites $31
    lis $31 ; Use $31 since we'll restore it
    .word 4
    add $30, $30, $31
    lw $31, -4($30) ; Pop $31 from stack
    jr $31 ; Return to loader
```

- Procedure Changes

```
f:
    sw $1, -4($30) ; Push registers we will modify
    sw $2, -8($30)
    lis $2
    .word 8
    sub $30, $30, $2 ; Decrement stack pointer
    ; Insert procedure here
    add $30, $30, $2 ; Assuming $2 is still 8
    lw $2, -8($30) ; Pop registers to restore
    lw $1, -4($30)
    jr $31 ; New line!
```

Lecture 5

- Note: there is NO default value to register, so remember to initialize it
- How do we pass parameters?

```
void f(int a, int b) {}
```

- Example: sumEvens1ToN adds all even numbers from 1 to N
  - `$1` Scratch Register (Should Save!)
  - `$2` Input Register (Should Save!)
  - `$3` Output Register (Do NOT Save!)

```
; The idea is:
; $3 = 0
; $1 = $2 % 2
; $2 = $2 - $1
; $1 = 2
; Top: $3 = $3 + $2
;      $2 = $2 - 2
;      if ($2 != 0) go to top
; jr $31

lis $1
.word 8
sw $1, -4($30)
sw $2, -8($30)
sub $30, $30, $1
add $3, $0, $0
lis $1
.word 2
div $2, $1
mfhi $1
sub $2, $2, $1
lis $1
.word 2
top:
add $3, $3, $2
sub $2, $2, $1
bne $2, $0, top
lis $1
.word 8
add $30, $30, $1
lw $2, -8($30)
lw $1, -4($30) ; Reload $1 and $2
jr $31 ; Back to caller
```

- Input and Output
  - We do this one byte at a time!
  - Output: Use `sw` to store words in location `0xffff000c`. Least significant byte will be printed.
  - Input: Use `lw` to load words in location `0xffff0004`. Least significant byte will be the next character from stdin.
  - Input/Output the ASICC character

```
lis $1
.word 0xffff000c
lis $2
.word 48 ; In ASCII code, 48 means 0!
sw $2, 0($1)
```

- Example: Printing CS241 to the screen followed by a newline character:

```
lis $1
.word 0xffff000c
lis $2
.word 67 ; C
sw $2, 0($1)
lis $2
.word 83 ; S
sw $2, 0($1)
lis $2
.word 50 ; 2
sw $2, 0($1)
lis $2
sw $2, 0($1)
lis $2
.word 49 ; 1
sw $2, 0($1)
lis $2
.word 10 ; \n
sw $2, 0($1)
jr $31
```

- Part of our long-term goal is to convert assembly code (our MIPS language) into machine code (bits).
  - Input: Assembly code
  - Output: Machine code
- Any such translation process involves two phases: Analysis and Synthesis.
  - Analysis: Understand what is meant by the input source
  - Synthesis: Output the equivalent target code in the new format

- What if a label is used before it is defined? We don't know the address when it's used!
  - Perform two passes:
    - Pass 1: Group tokens into instructions and record addresses of labels (data structure?).
    - Note: multiple labels are possible for the same line! For example, `f: g: add $1, $1, $1`.
    - Pass 2: translate each instructions into machine code. If it refers to a label, look up the associated address compute the value.
- A label at the end of code is allowed (it would be the address of the first line after your program).
- Our instruction (`bne`) can be broken down as follows:

- Only the offset part `i` is signed, others are unsigned. That's why we need to do bit masking for `i` with `0xFFFF` (We do not want the leading 1 will overwrite our results if `i` is negative)!
- We can use bit shifting to put information into the correct position, and use a bitwise or to join them:

```
int instr = (5 << 26) | (2 << 21) | (0 << 16) | offset
```

- Recall in C++, ints are 4 bytes. We only want the last two bytes. First, we need to apply a "mask" to only get the last 16 bits:

```
offset = -3 & 0xffff
```

- Printing Bytes in C++

```
int instr = (5 << 26) | (2 << 21) | (0 << 16) | (-3 & 0xffff);
unsigned char c = instr >> 24;
cout << c;
c = instr >> 16;
cout << c;
c = instr >> 8;
cout << c;
c = instr;
cout << c; // will output the least sinificant 8 bits
```

- Note: You can also mask here to get the 'last byte' by doing `& 0xff` if you're worried about which byte will get copied over.

Lecture 6

- Definition: An alphabet is a non-empty, finite set of symbols, often denoted by Σ (capital sigma).
- Definition: A string (or word) w is a finite sequence of symbols chosen from Σ. The set of all strings over an alphabet Σ is denoted by Σ^*.
- Definition: A language is a set of strings.
- Definition: The length of a string w is denoted by $|w|$.
- Since an alphabet is a set of "symbols" (which is vague), and a language is a set of words, a language can be the alphabet of another language
- Examples - Alphabets:
  - $\Sigma = a, b, c, \ldots, z$, the Latin (English) alphabet.
  - $\Sigma = 0, 1$, the alphabet of binary digits.
- Examples - Strings:
  - ϵ (epsilon) is the empty string. It is in Σ^* for any Σ. $|\epsilon| = 0$
  - For $\Sigma 0, 1$, strings include $w = 011101$ or $x = 1111$. Note $|w| = 6$ and $|x| = 4$.
  - For our course, assume Σ will never contain the symbol ϵ. ϵ is just a notational convention; the actual string is empty.
- Examples - Languages:
  - $L = \emptyset$ or , the empty language
  - $L = \epsilon$, the language consisting of (only) the empty string
- Why are finite languages are easy to determine membership?
  - To determine membership in a language, just check for equality with all words in the language!
- Definition: A regular language over an alphabet Σ consists of one of the following:
  - The empty language and the language consisting of the empty word are regular.
  - All languages a for all $a \in \Sigma$ are regular.
  - The union, concatenation or Kleene star (pronounced klay-nee) of any two regular languages are regular.
  - Nothing else.

- Basically, if L is finite, L is regular.
- Let L, L_1 and L_2 be three regular languages. Then the following are regular languages
  - Union: $L1 \cup L2 = x : x \in L_1 \text{ or } x \in L_2$
  - Concatenation: $L_1 \cdot L_2 = L_1 L_2 = xy : x \in L_1, y \in L_2$
  - Kleenestar:$L^* = \epsilon \cup xy : x \in L^*, y \in L = \cup_{n=0}^{\infty} L^n$
- Error States in CS 241: if a bubble does not have a valid arrow leaving it, we assume this will transition to an error state.
- Definition: A DFA(Deterministic Finite Automata) is a 5-tuple $(\Sigma, Q, q0, A, \delta)$:
  - Σ is a finite non-empty set (alphabet).
  - Q is a finite non-empty set of states.
  - $q0 \in Q$ is a start state
  - $A \subseteq Q$ is a set of accepting states
  - $\delta : (Q \times \Sigma) \rightarrow Q$ is our [total] transition function (given a state and a symbol of our alphabet, what state should we go to?).
- Rules for DFAs
  - States can have labels inside the bubble. This is how we refer to the states in Q.
  - For each character you see, follow the transition. If there is none, go to the (implicit) error state.
  - Once the input is exhausted, check if the final state is accepting. If so, accept. Otherwise reject.

Lecture 7

- We can extend the definition of $\delta : (Q \times \Sigma) \rightarrow Q$ to a function defined over $Q \times \Sigma^*$ via:

$$\delta^* : (Q \times \Sigma^*) \rightarrow Q$$

$$(q, \epsilon) \rightarrow q$$

$$(q, aw) \rightarrow \delta^*(\delta(q, a), w)$$

- where $a \in \Sigma$ and $w \in \Sigma^*$ (aw is concatenation). Basically, if processing a string, process a letter first then process the rest of a string.
- Defitnion: A DFA given by $M = (\Sigma, Q, q_0, A, \delta)$ accept a string w if and only if $\delta^*(q_0, w) \in A$.
- Defintion: The language of a DFA M is the set of all strings accepted by M, that is:

$$L(M) = w : M \text{ accepts } w$$

- Theorem (Kleene): L is regular if and only if $L = L(M)$ for some DFA M. That is, the regular languages are precisely the languages accepted by DFAs.
- Implementing a DFA

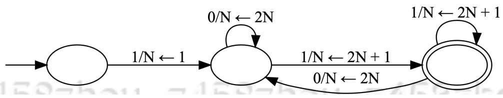
Algorithm 2 DFA Recognition Algorithm

w = $a_1 a_2 .... a_n$
$s = q_0$
for i in 1 to n **do**
    s = $\delta(s, a_i)$
end for
if $s \in A$ **then**
    Accept
else
    Reject
end if

- Example:

We could also have DFAs where we attach actions to arcs.

- For example, consider a subset of the language of binary numbers without leading zeroes described below.
- We'll create a DFA where we also compute the decimal value of the number simultaneously. Could then print the value.
- Look at the DFA corresponding to 1(0 | 1)*1.
- In what follows, you should read 1/N ← 2N + 1 as: the leftmost 1 corresponds to a DFA transition, the / has no meaning, and the N ← 2N + 1 changes N to be 2N + 1.




```
w = 1011
(1) s = q0 -> q1, N = 1
(2) s = q1, 0 ->q1, N = 2*N = 2
(3) s= q1, 1 -> q2, N = 2N + 1 = 2*2+1 = 5
(4) s = q2, 1 -> q2, N = 2N + 1 = 2*5+1 = 11
```

- When we allow for a state to have multiple branches given the same input, we say that the machine "chooses" which path to go on. To make the right choice, we would need an oracle that can predict the future, so to actually implement this, we would need to try every choice (yuck!)
  - This is called non-determinism.
  - We then say that a machine accepts a word w if and only if there exists some path that leads to an accepting state!
  - We can then simplify the previous example to an NFA as defined on the next point.
- Definition: Let M be an NFA. We say tha M accepts w if and only if there exists some path through M that leads to an accepting state. The language of an NFA M is the set of all strings accepted by M, that is:

$$L(M) = w : M \text{ accepts } w$$

- Definition: A NFA(Non-Deterministic Finite Automata) is a 5-tuple $(\Sigma, Q, q_0, A, \delta)$:
  - Σ is a finite non-empty set (alphabet).
  - Q is a finite non-empty set of states.
  - $q_0 \in Q$ is a start state
  - $A \subseteq Q$ is a set of accepting states
  - $\delta : (Q \times \Sigma) \to 2^Q$ is our [total] transition function. Note that 2^Q denotes the power set of Q, that is, the set of all subsets of Q. This allows us to go to multiple states at once!
- Again we can extend the definition of $\delta : (Q \times \Sigma) \to 2^Q$ to a function $\delta^* : (2^Q \times \Sigma^*) \to 2^Q$ via:

$$\delta^* : (2^Q \times \Sigma^*) \to 2^Q$$

$$(S, \epsilon) \mapsto S$$

$$(S, aw) \mapsto \delta^* \left(\bigcup_{q \in S} \delta(q, a), w \right)$$

where $a \in \Sigma$.

- Definition: An NFA given by $M = (\Sigma, Q, q_0, A, \delta)$ accepts a string w if and only if $\delta^*(\{q_0\}, w) \cap A \neq \emptyset$
- Simulating an NFA

Algorithm 3 NFA Recognition Algorithm

$w = a_1 a_2 ... a_n$
$S = \{q_0\}$
for i in 1 to n **do**
  $S = \bigcup_{q \in S} \delta(q, a_i)$
end for
if $S \cap A \neq \emptyset$ **then**
  Accept
else
  Reject
end if

- NFAs are not more powerful than DFAs!
  - Why not: Even the power-set of a set of states is still finite. So, we can represent sets of states in the NFA as single states in the DFA!
  - NOTE: We are about to go over the algorithm for how to do this conversion. The algorithm itself is *optional* material, but you should understand the concept above.

Lecture 8

- From Kleene's theorem, the set of languages accepted by a DFA are the regular languages
- The set of languages accepted by DFAs are the same as those accepted by NFAs
- Therefore, the set of languages accepted by an NFA are precisely the regular languages!
- Definition: A ε-NFA(ε-Non-Deterministic Finite Automata) is a 5-tuple $(\Sigma, Q, q_0, A, \delta)$:
  - Σ is a finite non-empty set (alphabet) that does not contain the symbol ε.
  - Q is a finite non-empty set of states.
  - $q_0 \in Q$ is a start state
  - $A \subseteq Q$ is a set of accepting states
  - $\delta : (Q \times \Sigma \cap \{\varepsilon\}) \to 2^Q$ is our [total] transition function. Note that 2^Q denotes the power set of Q, that is, the set of all subsets of Q. This allows us to go to multiple states at once!
- These ε-transitions make it trivial to take the union of two NFAs.

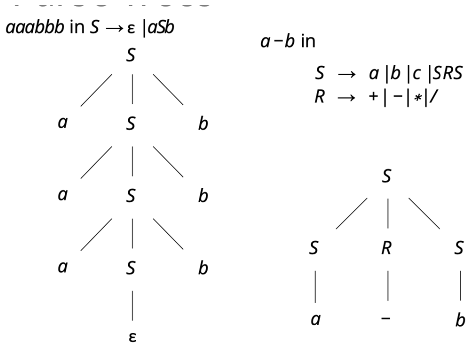
Lecture 9

- Maximal Munch: Consume characters until you no longer have a valid transition. If you have characters left to consume, backtrack to the last valid accepting state and resume.
  - We only backtrack exactly once in this course, since once we backtrack, we output a token, and we cannot take this action back.
  - It is of course possible to backtrack more, using a stack and buffering the output, but we choose not to do this in this course (or in most practical implementations).
- Simplified Maximal Munch: Consume characters until you no longer have a valid transition. If you are currently in an accepting state, produce the token and proceed. Otherwise go to an error state.
- Definition: Grammar is the language of languages (Matt Might).
  - In some sense, grammars help us to describe what we are allowed and not allowed to say.
  - Context-free grammars are a set of rewrite rules that we can use to describe a language.
- Definition: A Context Free Grammar (CFG) is a 4 tuple (N, Σ, P, S) where
  - N is a finite non-empty set of non-terminal symbols.
  - Σ is an alphabet; a set of non-empty terminal symbols.
  - P is a finite set of productions,each of the form $A \to \beta$ where $A \in N$ and $\beta \in (N \cup \Sigma)^*$
  - $S \in N$ is a start symbol
- Note: we set $V = N \cup \Sigma$ to denote the vocabulary, that is, the set of all symbols in our language.
- Convention:
  - Lower case letters from the start of the alphabet, i.e., $a, b, c, ...$, are elements of Σ
  - Lower case letters from the end of the alphabet, i.e., w, x, y, z, are elements of Σ^* (words)
  - Upper-case letters, i.e., A, B, C, ..., are elements of N (non-terminals)
  - S is always our start symbol.
  - Greek letters, i.e., α, β, γ, ..., are elements of $V^$(recall this is*(N \cup \Sigma)*^$)
- Defintion: Define the language of a CFG (N, Σ, P, S) to be $L(G) = w \in \Sigma^* : S \Rightarrow^* w$.
- Definition: A language is context-free if and only if there exists a CFG G such that $L = L(G)$.
- Every regular language is context-free!

Lecture 10

- Warm up problem: write a CFG that recognizes $L = \{a^i b^j c^i : i, j \in N\}$
  - $S \to aSc|T, T \to bT|\epsilon$
- Leftmost derivation: In the first derivation, we chose to do a left derivation, that is, one that always expands from the left first.
- Rightmost derivation: In the second derivation, we chose to do a right derivation, that is, one that always expands from the right first.
- Parse trees
  - Note: To every left- (right-) most derivation there exists a unique parse tree (and vice versa).
  - A derivation uniquely defines a parse tree.
  - An input string can have more than one parse tree.

- Leaf nodes represent terminals or ε. It's common to exclude the ε leaf node for ε productions, in which case a non-terminal can also be a leaf node, but only if its expansion was ε.



- Is it possible for multiple leftmost derivations (or multiple rightmost derivations) to describe the same string?
    - Yes
- Defintion: A grammar for which some word has more than one distinct leftmost derivation/rightmost derivation/parse tree is called ambiguous.
- Right Associative Grammar: $S \to LRS|L$; Left Associative Grammar: $S \to SRL|L$.
    - We can make a grammar left/right associative by insisting on how the recursion works
- Recall, with our depth-first traversal, the deeper parts of the tree will be evaluated first therefore get a higher precedence

Lecture 11

- Top-Down Parsing
    - Start with S (the start symbol), and look for a derivation that gets us closer to w. Then, repeat with remaining non-terminals until we're done.
- Algorithm
    - note: broken even with faerie magic!

Algorithm Top-Down Parsing Algorithm (with faerie magic)

```
push S
for each 'a' in input do
    while top of stack is A ∈ N do
        pop A
        ask a magic faerie to tell you which production A →γ to use
        push the symbols in γ (right to left)
    end while
    // TOS is a terminal
    if TOS is not 'a' then
        Reject
    else
        pop 'a'
    end if
end for
Accept
```

- When we reached the end of the input, we had no way of realizing we weren't done with the stack
    - We can make sure we recognize the end (and beginning) of the file by giving them symbols ⊢ and ⊣ such that $S' \to \vdash S \dashv$
    - ⊢ and ⊣ are terminals
- Algorithm

Algorithm Top-Down Parsing Algorithm (with faerie magic)

```
push S
for each 'a' in input do
    while top of stack is A ∈ N do
        pop A
        ask a magic faerie to tell you which production A →γ to use
        if there is a valid production A →γ then
            push the symbols in γ (right to left)
        else
            reject
        end if
    end while
    // TOS is a terminal
    if TOS is not 'a' then
        Reject
    else
        pop 'a'
    end if
end for
Accept
```

- Predictor table
    - Good news: can have descriptive error messages (expected x, found y)

    - Bad news: need to always know what to do based on one symbol
    - Ugly news: needed to transform our grammar to make it fit, not all grammars can be transformed, and for those that can, we change the parse trees!
- Definition: A grammar is called $LL(1)$ if and only if each cell of the predictor table contains at most one entry.
- Why it's called $LL(1)$?
    - First L: Scan left to right (more precisely,beginning to end)
    - Second L: Left most derivations
    - Number of symbol look ahead: 1

Lecture 12

- Definition: Predict(A, a) = $\{A \to \beta : a \in$ First(β) $\cup$ $\{A \to \beta :$ Nullable(β) and $a \in$ Follow(A)$\}\}$
- Note that Nullable(β) = false whenever β contains a terminal symbol.
- Further, Nullable(AB) = Nullable(A) $\wedge$ Nullable(B)
- Thus, it suffices to compute Nullable(A) for all $A \in N$
- Computing Nullable

Algorithm 2 Nullable(A) for all $A \in N'$

```
Initialize Nullable(A) = false for all A ∈ N'.
repeat
    for each production in P do
        if (P is A → ε) or (P is A → B_1 ··· B_k and ⋀_{i=1}^{k} Nullable(B_i) = true) then
            Nullable(A) = true
        end if
    end for
until nothing changes
```

- Computing First

Algorithm 3 First(A) for all $A \in N'$

```
Initialize First(A) = {} for all A ∈ N'.
repeat
    for each rule A → B_1 B_2 ··· B_k in P do
        for i ∈ {1, .., k} do
            if B_i ∈ T' then
                First(A) = First(A) ∪ {B_i}; break
            else
                First(A) = First(A) ∪ First(B_i)
                if Nullable(B_i) == False then break
            end if
        end for
    end for
until nothing changes
```

- Computing First (2)

Algorithm 4 First(β) where $\beta = B_1 \cdots B_n \in V^*$

```
result = ∅
for i ∈ {1, .., n} do
    if B_i ∈ T' then
        result = result ∪ {B_i}; break
    else
        result = result ∪ First(B_i)
        if Nullable(B_i) == False then break
    end if
end for
```

Lecture 13

- Defintion: we say that a $\beta \in V^*$ is nullable if and only if Nullable(β) = true.
- Defintion: $Predict(A, a) = \{A \to \beta : a \in First(\beta)\} \cup \{A \to \beta : Nullable(\beta) \text{ and } a \in Follow(A)\}$
- Notice that this still requires that the table only have one member of the set per entry to be useful as a deterministic algorithm.
- Computing Follow

Algorithm 5 Follow(A) for all $A \in N$ (Recall N = $N' \setminus \{ S' \}$)

```
Initialize Follow(A) = {} for all A ∈ N.
repeat
    for each production A → B_1 B_2 ··· B_k in P' do
        for i ∈ {1, .., k} do
            if B_i ∈ N then
                Follow(B_i) = Follow(B_i) ∪ First(B_{i+1} ··· B_k)
                if ⋀_{m=i+1}^{k} Nullable(B_m) == True or i == k then
                    Follow(B_i) = Follow(B_i) ∪ Follow(A)
                end if
            end if
        end for
    end for
until nothing changes
```

- Computing Predict

Algorithm 6 Computing Predict

Initialize Predict[A][a] $= \{\}$ for all $A \in N'$ and $a \in T'$.
for each production $A \rightarrow \beta$ in P **do**
  for each $a \in First(\beta)$ **do**
    Add $A \rightarrow \beta$ to Predict[A][a]
  end for
  if Nullable(β) == True **then**
    for each $a \in Follow(A)$ **do**
      Add $A \rightarrow \beta$ to Predict[A][a]
    end for
  end if
end for

- Summary and Examples
  - Nullable:
    - `A → ε` implies that `Nullable(A)` is true.
      Further `Nullable(ε)` is true.
    - If `A → B1...Bn` and each of `Nullable(Bi)` is true then `Nullable(A)` is true.
  - First:
    - `A → aα` then $a \in First(A)$
    - `A → B1...Bn` then $First(A) = First(A) \cup First(Bi)$ for each $i \in 1, \ldots, n$ until `Nullable(Bi)` is false. One stills needs to include $First(Bi)$ in $First(A)$ for the first $Nullable(B_i) = False$ and terminate.
  - Follow:
    - `A → aBβ` then $Follow(B) = First(\beta)$
    - `A → aBβ` and `Nullable(β)` is true, then $Follow(B) = Follow(B) \cup Follow(A)$
  - Predict:
    - $Predict(A, a) = A \rightarrow \beta \mid a \in First(\beta) \cup A \rightarrow \beta \mid \beta$ is nullable and $a \in Follow(A)$
- Based on the definitions of Nullable, First and Follow, we can conclude that a grammar is LL(1) if and only if:
  - no two distinct rules with the same LHS can generate the same first terminal
  - no nullable symbol A has the same terminal a in both its first and follow sets
  - there is only one way to derive ε from a nullable symbol
- Left recursive grammars are never LL(1). A left recursive grammar, as the name indicates, is a grammar where the recursion on a non-terminal happens on the left within the right-hand side. For example, the rule S → S + T is left recursive, the S symbol within S + T is on the left. A Right Recursive rule would take the form S → T + S.
- A grammar with two or more rules for the same non-terminal with a common left prefix of length k, cannot be LL(k).
- If $A \rightarrow \alpha\beta_1 | \cdots |\alpha\beta_n|\gamma$ where $\alpha \neq \epsilon$ and γ is representative of other productions that do not begin with α, then we can change this to the following equivalent grammar by left factoring (a grammar has rules $A \rightarrow \alpha\beta_1 | \cdots |\alpha\beta_n$, all with a common prefix $\alpha \neq \epsilon$ on the right hand side):

```
A -> αB|γ
B -> β1| ...|βn
```

- An alternate approach to make a [direct] left recursive grammar right recursive; say `A → Aα|β` where β does not begin with the non-terminal A, we remove this rule from our grammar and replace it with:
  - A -> βA'
  - A' -> αA'|ε
- The Top-Down parsing algorithm we have studied, LL(1), is not compatible with left-associative grammars, or with grammars with left-ambiguous productions. The right associative grammar would require the programmer to introduce parentheses to force the left association, e.g. (x-y)+z.
- Bottom-Up Parsing idea: look for the RHS of a production, replace it with the LHS. When you don't have enough for a RHS, read more input. Keep grouping until you reach the start state.
  - Our invariant here will be Stack + Unread Input = αi . (Contrast to top-down where invariant was consumed input + reversed Stack contents = αi .)
- Theorem (Knuth 1965): For any grammar G, the set of viable prefixes (stack configurations), namely {αa: α ∈ V* is a stack, a ∈ Σ is the next character, ∃ x∈ Σ* such that S ⇒* αax} is a regular language, and the NFA accepting it corresponds to items of G! (Recall V = N ∪ Σ). Converting this NFA to a DFA gives a machine with states that are the set of valid items for a viable prefix!

Lecture 14

- Definition: An item is a production with a dot · somewhere on the right hand side of a rule.
  - Items indicate a partially completed rule.
  - That dot is called the "bookmark".
- We say that a grammar is LR(0) if and only if the LR(0) automaton does not have any shift-reduce or reduce-reduce conflicts.

Section 2: Tutorials

Tutorial 1

- What is Binary?
  - Binary - ways our machines encode info, and $b \in 0, 1$
- Eg. what is 1000 be
  - $2^3 = 8$ unsigned magnitude
  - -8 2's complement
  - $[T, F, F, F]$ array of bools
  - "backspace char" in ASCII
  - **Representation matters**
- Unsigned binary: n-bit binary number is represented as $b_{n-1}, b_{n-2}, \ldots, b_0$, where $b \in 0, 1$
- To convert to decimal: $2^{n-1} \times b_{n-1} + \ldots + 2^0 \times b_0$
- Decimal to binary
  - Idea 1: take the highest powers of 2 from the decimal (inefficient)
  - Idea 2: Repeatedly divide the number by 2, tracking the quotient & remainder
  - Eg. Convert 23 to binary

Number	Quotient	Remainder
23/2	11	1
11/2	5	1
5/2	2	1
2/2	1	0
1/2	0	1

    - and read from bottom to up, that will be $10111_2 = 23_{10}$
- 2's complement
  - Range of values for n-ary
    - Unsigned Binary $0 \sim 2^n - 1$
    - 2's complement $-2^{n-1} \sim 2^{n-1} - 1$
- Convert decimal to 2's complement
  - if number is ≥ 0: use the unsigned representation
  - if number is < 0:
    - Get the binary rep of the positive number
    - flip the bits
    - then add 1
- Convert from 2's complement to decimal
  - Method 1: if $b_{n-1} = 1$, then flip the bits, add 1 and negate the positive decimal
  - Method 2: Treat $b_{n-1} - > -2^{n-1} b_{n-1}$, and add the rest as unsigned representation
- Assembly
  - CS 241: MIPS
    - Runs programs $ stores its data all in MEM (RAM)
    - 32 bits system where instructions are encoded as 4 bytes (1 word)
    - Registers hold 1 word of info
    - Special registers
      - 0 = 0$, immutable
      - 31$, end address in RAM, `jr $31` means return address
      - 3, 29,30$
      - $iii \ldots$ <- 2's complement number
        - `divu, multu, addi ...` treats the register values as unsigned binary
  - **Programs live in the same space in MEM (RAM) as the data they operate on**
    - PC cannot distinguish the two
  - Fetch-Execute Cycle

```
PC = 0x00
while True do:    // until PC = $31
    IR = MEM[PC]
    PC = PC + 4
    ... decode & execute IR ...
done
```

- Constant Values
  - Use the Load Immediate Skip command (`lis $s`) followed by an instruction to save into `$s`.
    - `lis $s` -> skip the next instruction -> store that instruction into `$s`
    - Use with `.word i` to store `i` into `$s`
    - Eg. Store 10 into `$5`

```
lis $5
.word 10
; the above two lines of commands skips the .word 10 & sets $5 = 10
```

  - Machine Code (based on the order of machine code horizontally for `lis $5`)
    - 000000 (operating code)
    - 00000 (`$s`)
    - 00000 (`$t`)
    - 00101 (`$d`)
    - 00000 (dead code, always be 0)
    - 010100 (function code)
  - Machine Code (`.word 10`)
    - `00000 ....... 000 001010`

Tutorial 2

- Recall: Fetch-Execute Cycle

```
PC = 0x00
while PC != $31 do
    IR = MEM[PC]
    PC = Pc + 4
    ... run IR's command ...
done
```

- Loops
  - Use `bne`/`beq`
  - such that, `bne $s, $t, i` and `beq $s, $t, i`, and $PC = PC + 4 * i$
  - i = 2's complement or label
- eg. write a program that a ≥ 0 # n in `$1` and store $n!$ into `$3`

```
lis $3
.word 1
loop: beq $1, $0, end
      mult $3, $1
      mflo $3 ; $3*$1
      lis $11
      .word 1
      sub $1, $1, $11
      beq $0, $0, loop
end: jr $31
```

MIPS Array

- mips.array
  - `$1` = address of the start of your array `Arr`
  - `$2` = length of `Arr`
- eg. `Arr = [1,2,3]`
  - `$1` $= 0 \times$ ..., `$2`$= 3$
  - element 1: `MEM[$1]`, element 2: `MEM[$1 + 4]`, element 3: `MEM[$1 + 4*2]`
  - the end of the array is `$1 + 4 * $2`
- eg. write a program that returns the product of all elements in `$1` to `$3`

```
lis $1
.word 1
lis $4
.word 4
mult $2, $4
mflo $2 ; $2 = $2 * 4
add $2, $1, $2 ; $2 = $1 + 4 * $2
loop: beq $1, $2, end
      lw $5, 0($1) ; Arr[i] = *Arr
      mult $3, $5
      mflo $3 ; $3 = $3 * $5
      add $1, $1, $4 ; i++
      beq $0, $0, loop
end: jr $31
```

Stack

- `$30` = Stack Pointer
- Initially, out of bounds address (since it is very last of the memory)
- grow backwards in MEM
- Idea: Preservation
  - Ensures the user/client that only the expected registers are mutated
- Push

```
sw $1, -4($30)
lis $1
.word $4
sub $30, $30, $1
.
. code program
.
```

- Pop

```
.
. code program
.
lis $1
.word 4
add $30, $30, $1
lw $1,  -4($30) ; $1 = old value
```

- Eg. write the factorial program (eg 1) but ensure all registers aside from $3 are preserved

```
sw $1, -4($30)
sw $11, -8($30)
sw $4, -12($30)
list $4
.word 12
sub $30, $30, $4
.
. factorial code from the previous example
.
lis $4
.word 12
add $30, $30, $4
lw $1, -4($30)
lw $11, -8($30)
lw $4, -12($30)
jr $31
```

Procedures & Recursion

- Procedure ≈ function names
  - Label with code & a `jr` often represent our function area
  - return value is equivalent to `$3`
- Recursion
  - Stack & `jalr`
  - `jalr $s`: `$31`=PC, PC=`$s`
  - Push `$31` & all parameters
  - `jalr` procedure
  - Popping `$31`

- eg. factorial with recursion

```
fact:
sw $1, -4($30)
sw $11, -8($30)
sw $31, -12($30)
lis $31
.word 12
sub $30, $30, $31

lis $11
.word 1

bne $1, $0, recur
add $3, $11, $0 ; base case: Set $3 = 1 & unwind
beq $0, $0, unwind ; base case

recur: ; call fact($1 - 1)
sub $1, $1, $11 ; $1 = $1 - 1
lis $31
.word fact
jalr $31 ; return to from the base case
add $1, $1, $11 ; return here after jr $31, restore old $1
mult $3, $1 ; Multiply previous answer by $1 to get new factorials
mflo $3

unwind:
lis $31
.word 12
add $30, $30, $31
lw $1, -4($30)
lw $11, -8($30)
lw $31, -12($30)
jr $31 ; jump to jalr or terminates
```

Tutorial 3

Symbol Tables

- Assembly is divided into 2 phases
  - Analysis
    - Check inputs & instruction syntax correctness
    - Construct the symbol table
    - Symbol table
      - Stores the values of all defined labels in code (`foo:`)
  - Synthesis: use the symbol table to:
    - Substitute labels with their values (`.word foo`)
    - Compute branch offsets (`bne`/`beq`)
- 2-phases are needed because of label definitions & substitutions
- 2 passes:
  - 1st pass - construct the symbol table
    - check for duplicated label definitions
  - 2nd pass
    - check for undefined labels
  - Check the label address falls within the range of your programs
- eg. Construct the symbol table for

```
begin:
label: beq $0, $0, after
jr $4
after:
sw $31, 0($30)
lis $4
abcO: ab1: .word begin
```

- Solution: label values = number of lines of non-null instructions *preceeding the label definition* $\times 4$
  - Symbol table

Number	Quotient
begin	$0 \times 4 = 0$

Number	Quotient
label	$0 \times 4 = 0$
after	$2 \times 4 = 8$
abcO	$4 \times 4 = 16$
ab1	$4 \times 4 = 16$

Generate Machine Code

- Goal: replicate cs241.binasm
- Ranges
  - Registers $\rightarrow \in [0, 31]$
  - `.word i`
    - $i = \text{decimal} \in [-2^{31}, 2^{31} - 1]$
    - $i = \text{hex} \in [0 \times 00000000, 0 \times ffffffff]$
    - $i = \text{label} \leftarrow \in \text{decimal}$
  - Immediates $(i) \rightarrow$ `sw/lw/bne/beq`
    - $i = \text{decimal} \in [-2^{15}, 2^{15} - 1]$
    - $i = \text{hex} \in [0 \times 0000, 0 \times ffff]$
    - $i = \text{label} \in \text{decimal}$

Outputing Binary

- Goal: cs241.binasm
  - doesn't output ASCII binary: input 2 does not output '00...10'
  - raw binary data
- eg. Write a program in C++ that takes in a singled integer (`.word`) & outputs its raw binay
  - Idea: `cout << char` $\rightarrow$ output binary of char
  - `sizeof(char)` = 1 byte = 8 bits
  - output 4 bytes with 4 chars
- Function to output word in C++

```
void output_word(int word) {
    char c; // 11110000 ......24......
    c = (word >> 24) & 0xff ; first 8 bits
    // word = 11110000 0101......0101
    // word >> 24 = 000...000(24 bits) 11110000, & 0xff ensures the first 24 bits are maksed with 0
    cout << c;
    c = (word >> 16) & 0xff ; second 8 bits
    cout << c;
    c = (word >> 8) & 0xff ; third 8 bits
    cout << c;
    c = (word) & 0xff ; fourth (last) 8 bits
    cout << c;
}
```

- Register Format

```
000000 sssss ttttt ddddd 00000 ffffff // ffffff is function code
```

- To print this out: `(s << 21) | (t << 16) | (d << 11) | f; s,t,d` $\in [0, 31]$.
- Immediate Format (`sw/lw/ bne/beq`)

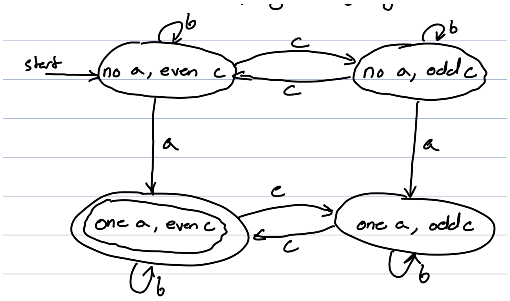
```
pppppp sssss ttttt iiiiiiii iiiiiiii // pppppp is opcode
```

- to print this out: `(p << 26) | (s << 21) | (t << 16) | (i & 0xffff)` // (i & 0xffff) <- 16 bits

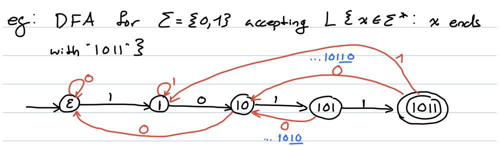
Tutorial 4

Regular Expression

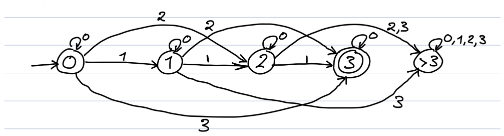
- Alphabet (Σ)
  - = finite sequence of symbols
- Word (w) = sequence of tokens $\in \Sigma$
- Example: $\Sigma = \{a, b, c\}$, w = 'a', 'ab', 'abc', $\epsilon \in \Sigma^*$
- Regular Languages (L) over some Σ with the iterative definition
  - $L = \emptyset$
  - $L = \{\epsilon\}$
  - $L = \{a\}$ where $a \in \Sigma$
  - $L = L_1 \cup L_2 = \{x | x \in L, \text{ or } x \in L_2\}$
  - L = L_1L_2 = \lbrace xy | x \in L_1 & y \in L_2 \rbrace
  - $L = L_1^* = \{\epsilon, L_1, L_1 L_1, \ldots, L_1 \ldots L_1\}$
- Regular Expression = concise definition of a recurrence language
  - $R = \emptyset$
  - $R = \epsilon$
  - $R = a, a \in \Sigma$
  - $R = R_1 | R_2, \{x | x \in R_1, \text{ or } x \in R_2\}$
  - R = R_1 R_2, \lbrace x | x \in R_1 & x \in R_2\rbrace
  - $R = R_1^* = ``", R_1, R_1 R_1, \ldots, R_1 \ldots R_1$
- Precendence of ops: $R^* > R_1 R_2 > R_1 | R_2$
  - $aa|bb^* \equiv (aa)|(b(b^*))$
- Example: regex for $\Sigma = \{a, b\}$ over $L = \{aa, ab, ba, bb\} \to aa, ab, ba, bb \in L, \epsilon, aaa \notin L, R = aa|ab|ba|bb = (a|b)(a|b)$
- Example: regex for \Sigma = \lbrace0,1 \rbrace, L = \lbrace w \in \Sigma^* | 2^{nd} \text{ symbol } = 0 & 5^{th} \text{ symbol } = 1 \rbrace, R = (0|1)0(0|1)(0|1)1(0|1)^*, which means second and fifth are fixed to 0 and 1 respectively
- Example: regex over $\Sigma\{a, b, +, -, /, \times\}, L = \{w \in \Sigma^* | w \text{ represents an arithmetic operation}\}$
  - Assume, no unary ($-b \notin L$), no implicit $\times (ab \notin L)$
  - term (a, b) op ($+, -, /, \times$) term
  - term op term op term op ...
  - $R = (a|b)[(+|-|/|\times)(a|b)]^*$ **DFAs**
- DFAs = models of computation
- $\{\Sigma, Q, q_0, A, \delta\}$
  - Σ = input alphabet
  - Q = set of states
  - $q_0 \in Q$, starting state
  - $a \subseteq Q$, accepting states
  - $\delta : Q \times \Sigma \to Q$, transition function
    - $\delta(q, a) = q'$, where q,q' \in Q & a \in \Sigma
    - if $q' = \epsilon$(DNE) then reject our input
- Example: DFA over $\Sigma = \{a, b, c\}$ accpets $L = \{x \in \Sigma^*\} | x \text{ contains one } 'a' | x \text{ has an even number of } 'c'$



- Example: DFA over $\Sigma\{0,1\}$, accepting $L = \{x \in \Sigma^* | x \text{ endswith “1011”}\}$



- Example: DFA over $\Sigma = \{0,1,2,3\}$ for $L = \{x \in \Sigma^* | x\text{'s digit sum} = 3\}$
    - $012", 3", 111" \in L, 0", 4", 2", 311", 333" \notin L$



Tutorial 5

NFAs

- Idea: Entend the DFA transition function $\delta(q, a)$ to produce a set of states we can transition to! Let's define multiple transitions from $q \in Q$ on $a \in \Sigma$.
- In the definition of NFA (only the fifth point is different from that of DFA), 2^Q $\equiv$ Power set of Q $\equiv$ sets of all all subsets of Q
- In an NFA, we can traverse through multiple states at once! For $w \in \Sigma^*$
    - Collect a set of all possible states S we could be for all input consumed
        - start with $S = q_0$
        - For input $w_i \in w$, let $S' = \delta(q', w_i) | \forall q' \in S$ (new set where w_i is applied to all states in S) then set $S = S'$.
        - Accept $w \in \Sigma^*$ if at least 1 of the states $q' \in S$ is accepting else reject.

ϵ - NFAs

- Identical to NFAs, but we allow for transitions on ϵ
    - $\delta : Q \times (\Sigma \cup \epsilon)$
- Updates to S on $\forall w_i \in w$ can follow any number of ϵ-transitions
    - ϵ-closure: set of all states we can reach from S by following ϵ-transitions
- ϵ-transitions act like a union of the 2 languages
    - $L \equiv L_1 \cup L_2$, ϵ-transitions let us match both `L_1 & L_2` simultaneously.

Converting an NFA into a DFA

- Challenge: Convert NFA's multi-state $\delta : Q \times \Sigma \to 2^Q$ into DFA's single state $\delta : Q \times \Sigma \to Q$
- Idea: let DFA's Q be all possible NFA 2^Q
    - i.e.: $Q = 2^Q$ for all NFA transitions
- Formally, let NFA $N = (\Sigma, Q, q_0, A, \delta)$, our DFA $D = (\Sigma', Q', q_0', A', \delta')$ should be defined in terms of N as:
    - $\Sigma' = \Sigma$ (unchanged alphabet)
    - $Q' = 2^Q$ (the set of all subsets of Q)

- $q_0' = q_0$
- $A' = S \in 2^Q | S \cap A \neq \emptyset$
  - All sets in 2^Q where at lest one of the states is an accpeting state
- $\delta' : 2^Q \times \Sigma \to 2^Q$ (1 to 1 output! Deterministic!!!)

- NFA to DFA Conversion Algorithm:
  - Create a transition table: Rows = DFA states and Columns = $a \in \Sigma$
    1. Start with row q_0 and all columns empty
    2. For all $a \in \Sigma$, and all empty rows, fill in their transitions $\delta'(S, a)$
    3. Add any unique transitions from step 2 as new empty rows
    4. Repeat steps 2 and 3 until no new rows can be added!

Scanning

- Goal: Take non-empty input w and split it into non-empty token sequences.
  - Input $w = w_1 + w_2 + ... + w_n$ where $w_1, ..., w_n \in L$ for $n > 0$
    - `w_i` are tokens
    - `+` denotes string concatenation
  - Word w can be scanned with respect to L if $\exists w = w_1 + ... + w_n$ for $n > 0$
    - Maximal Munch and Simplified Maximal Munch are scanning algorithms.
- Simplified Maximal Munch (SMM):
  - Run DFA accepting L with input w
  - If transition in state q on character a does not exist:
    - If q is accepting, output current token and reset DFA with remaining input
    - If q is not accepting, produce ERROR and exit
  - Consume input chars from w until ERROR or no more input.
- **Maximal Munch (MM)**:
  - Run DFA accepting L with input w
  - Backtrack input & DFA if transition on state q with char a does not exist to last seen accepting state
    - Output token & reset DFA if accepting state is found/reached
    - ERROR if backtracking didn't reach an accepting state
- Possible that MM and SMM might scan/accept different w

Tutorial 6

LL(1) Parsing

- Inputs: CFG $G = (N, \Sigma, P, S)$ and input string $x \in \Sigma^*$
- Goal: Find derivations from start symbol S to input string x (i.e., $S \Rightarrow^* x$)
  - Let α_i be derivation i from $S \Rightarrow ... \Rightarrow \alpha_i \Rightarrow ... \Rightarrow x, i \geq 0$ or $\alpha_i \in (N \cup \Sigma)^*$ (Origin: $A \in N, A \to \alpha_i \in P$)
  - Goal: find $S \Rightarrow \alpha_0 \Rightarrow \alpha_1 \Rightarrow ... \Rightarrow \alpha_n \Rightarrow x$ to prove $x \in L(G)$ or prove $x \notin L(G)$
    - $L(G)$ = "Language of the grammar G"
- LL(1) -> One symbol lookahead
  - Parser makes decision about what rule α_i to apply based on *only the next symbol* in the input
  - Note: LL(1) parsing doesn't always work due to this limitation!

Predict Table

- Used to determine if LL(1) parsing will work for G
- Let
  - $A \in N$
  - $a, b, c \in \Sigma$
  - $\alpha, \beta \in (N \cup \Sigma)^*$ <- strings of non-terminals & terminals
  - $A \to \alpha \in P$
- `Predict(A, a)` contains rule `A -> α` if:
  1. `α` can derive a string whose first symbol is `a`
     - e.g., `A -> aβ`
  2. `α` can derive `ε` and `a` could immediately follow `A` in the derivation
     - e.g., to get rid of `A`, `"AaB" =>* "εaB"`

- Formally:
  - `First(α) = { b | α =>* bβ for some β }`
  - `Follow(A) = { c | S' =>* αAcβ for some α, β }`
  - `Nullable(α) = true if α =>* ε, false otherwise`
  - `Predict(A, a) = { A -> x | (a ∈ First(α)) or (Nullable(α) and a ∈ Follow(A))}`
- If `Predict(A, a)` contains ≤ 1 rule for all pairs `(A, a)`, we say `G` is `LL(1)` and we can use `LL(1)` to parse `G`
- Tip: when computing the predict table for a grammar, compute in the order `Nullable`, `First`, then `Follow` (use algorithm from class or intuition!)

Section 3: Reviews

- Midterm review is presented as hand-written notes.