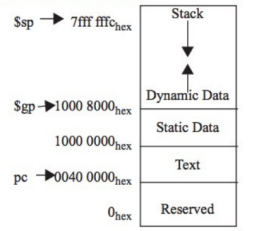
Assembly

To multiply 2 integers together:

```
li $t0, 5
li $t1, 7
mult $t1, $t0
mflo $t2
```

MEMORY ALLOCATION



What is

Preserved vs Unpreserved Regs

- **Preserved:** $s0 - $s7, and $sp , $ra
- **Unpreserved:** $t0 - $t9, $a0 - $a3, and $v0 - $v1

How do we get the negative of a value?
Invert all the bits (0→1, 1→0), and then add 1

What is the largest, most positive number you can put as an immediate? Ans: $2^{15} - 1$

```
# allocate 2 words on the stack
addiu $sp $sp -8   # <- address are unsigned
# e.g. store the values t0 and t1 on the stack
sw $t0 4($sp)
sw $t1 0($sp)

(here you can do whatever you want, as long as you put th
$sp back into the same spot)

# e.g. load the values t0 and t1 back off the stack
lw $t0 4($sp)
lw $t1 0($sp)
# unallocate 2 words from the stack
addiu $sp $sp 8   # <- remember the stack shrinks "up"
```

Loads versus Stores

- Load Word: `lw $t0 0($t1)`
  - Takes the address calculated as (0+$t1) and goes and finds the thing in that address and puts it into $t0. The value of $t0 is changed (to the thing found in memory at the specified address), while the value of $t1 is unchanged (think of $t1 as the pointer to a value in memory). This is also known as a "Memory Read".
- Store Word: `sw $t0 0($t1)`
  - Takes the value from $t0 and writes it out to memory at address (0+$t1). The values of $t0 and $t1 are both unchanged, the only thing that changes is the value of memory (at the specified address). This is also known as a "Memory Write".

- All 4 instructions are for getting a value into a register. But *where* they get the value from, and how the value is
- `li`
  - Used for taking an integer *constant* and loading it into a register
  - E.g. `li $t0, 5`
  - The value "5" is baked into the instruction in your program
- `la`
  - Used for taking a constant *label* and loading it into a register
  - E.g. `la $t0, MyVar`
  - The label "MyVar" is really just a name for *address*
- `move`
  - Take a register and copies the value to another register
  - E.g. `move $t0 $t1`
- `lw`
  - Used for getting a word (a 32-bit value) from memory at a particular address
  - E.g. `lw $t0 0($t1)`
  - $t1 would hold the address in memory (like a pointer) and the lw fetches the thing at that address and puts it

Instruction Representation in R-Type

op	rs	rt	rd	shamt	funct
6 b	5 b	5 b	5 b	5 b	6 b
31 – 26	25 – 21	20 – 16	15 – 11	10 – 6	5 – 0

- The combination of the **opcode** and the **funct** code tell the processor what it is supposed to be doing
- Example:

`add $t0, $s1, $s2`

op	rs	rt	rd	shamt	funct
0	17	18	8	0	32

op = 0, funct = 32 mean "add"
rs = 17 means "$s1"
rt = 18 means "$s2"
rd = 8 means "$t0"
shamt = 0 means this field is unused in this instruction

A full list of codes can be found in your MIPS Reference Card

I-Type Format

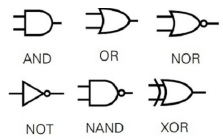
op	rs	rt	address
6 b	5 b	5 b	16 b
31 – 26	25 – 21	20 – 16	15 – 0

- The I-Type ***address*** field is a signed number
- The `addi` instruction is an I-Type, example:

```
addi $t0, $t1, 42
```

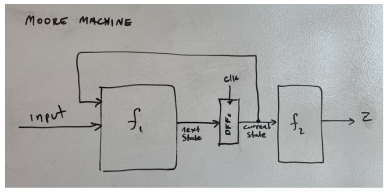
  - What is the largest, most positive, number you can put as an immediate? Ans: $2^{15} - 1$

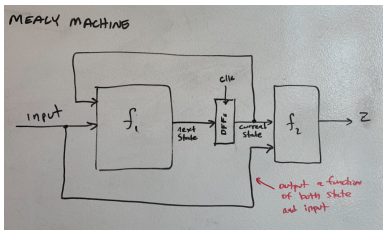
Digital Logic

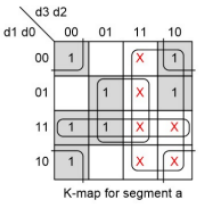


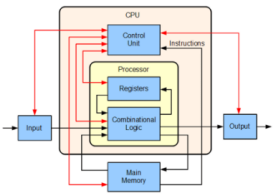
K-map Rules (reviewed)

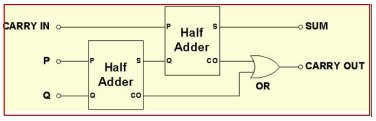
1. Groups can contain only 1s
2. Only 1s in adjacent groups are allowed
3. Groups may ONLY be horizontal or vertical (no diagonals)
4. The number of 1s in a group must be a power of two (1, 2, 4, 8...)
5. Groups must be as large AND as few in no.s as "legally" possible
6. All 1s must belong to a group, even if it's a group of one element
7. Overlapping groups are permitted
8. Wrapping around the map is permitted

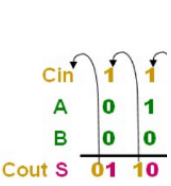




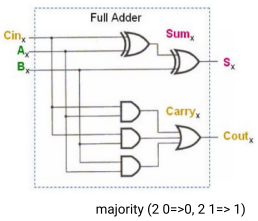






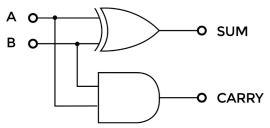


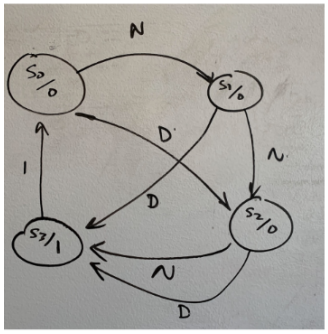
odd 0's => 0

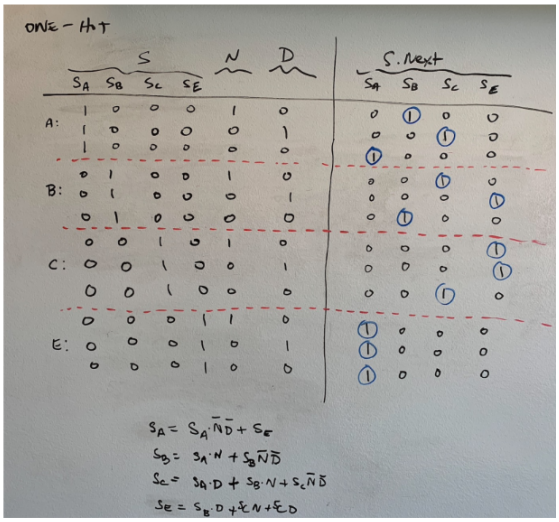


majority (2 0=>0, 2 1=> 1)

Half Adder







Digital Logic -> Assmebly

```
Initialize();                                    Pseudocode
while (true) {
    instruc_reg = GetFromMem[prog_countr];
    executeInstruc(instruc_reg);
    prog_countr++;
}
```

Instruction Register

```
li $t0, 5
```

Registers

```
$t0: 5
$t1: ?
$t2: ?
```

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.
Memory is addressed in Bytes (more on this later).

Program Counter

```
4
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t2, $t0, $t1
```

Arithmetic Logic Unit

```
0 + 4 = 4
```

Core Components

What we need in a CPU is:

- Some place to hold the statements (instructions to the CPU) as we operate on them → MEMORY
- Some *place* to tell us *which statement* is next → PROGRAM COUNTER (PC)
- Some *place* to hold the variables → REGISTERS
- Some *way* to do arithmetic on *numbers* → ARITHMETIC LOGIC UNIT (ALU)

...And one more thing:

- Some place to tell us which statement is currently being executed → INSTRUCTION REGISTER (IR)

Basic Interaction

- Copy instruction from memory at wherever the program counter (PC) says into the instruction register (IR)
- Execute it, possibly involving registers and the arithmetic logic unit (ALU)
- Update the PC to point to the next instruction
- Repeat