Alex Mei | CS64 | Winter 2021

CS 64 NOTES

Main Components of a Computer:

- Processor: execute program instructions
- Memory
- Input (Devices): keyboard, mouse, etc
- Output (Devices): screen, speaker, etc.
- Secondary Data Storage

Volatile: main memory is considered volatile in that it is wiped when turned off; secondary data storage is not volatile

Computer Memory: combination of address (location of data) and payload (actual data)

- smallest representation is a bit (0 or 1)
- 1 byte = 8 bits; 1 nibble = 4 bits
- input and output are all in bits (machine language)

Parts of the CPU:

- Arithmetic Logic Unit (ALU): does binary calculations using registers and logic circuits
- Control Unit (CU): interprets instructions intro control codes for ALU and memory

Fetch-Execute Cycle:

- Fetch the next instruction in the program
- Decode the instruction
- Get stored data as necessary
- Execute the instruction (and store new data as necessary)

Positional Notation: method to convert base to decimal

- rightmost digit is the 0th position = multiplier * base raised to the power of the position
- 2^{10} = 1024 = 1 kilo; 2^{20} = 1024 kilos = 1 mega; 2^{30} = 1024 megas = 1 giga

Machine vs Assembly:

- Machine Language (ML) is the bits
- Assembly Language is given in mnemonic codes displayed one step at a time (for better readability)
- Compiler: translates high level to low level languages (usually assembly)
- Assembler: translates assembly language to machine language

Speed affected by ordering of instructions, location in memory, instruction deconstruction, and order of pipeline

Digital Design: logical decisions made with bits

- Combinatorial Logic: and, or, xor, etc
- Sequential Logic: latches, ff, fsm, etc.

Conversions:

- Hexadecimal Conversion: A = 10, B = 11, C = 12, D = 13, E = 14, F = 15
- Binary to Octal: Gather 3 Method
- Binary to Hex: Gather 4 Method
- Octal/Hex to Binary: Reverse Gather
- To Decimal: Positional Notation
- Decimal To: Divisor Remainder Method

Notation:

- Binary: 0b
- Hex: 0x

Two's Complement: flip all the bits, and add one to negate

- must specify number of bits
- negative numbers have "1" as most significant number (positive numbers have "0")
- range: 0 to $2^n - 1$ (unsigned) or -2^{n-1} to $2^{n-1} - 1$ (two's complement)

Adding Binary:

- N Bit Adder needs 2N (two numbers) + 1 (carry in) bits for input and N (result) + 1 (carry out) for output
- Output Carry Bit (C): carry out at the most significant column due to computation limit (unsigned)
- Overflow (V): only for negative number addition (signed)
    - Either [x, y > 0 AND s < 0] OR [x, y > 0 AND s < 0] for x + y = s
    - s is the sign of the resulting signed number

Binary Logic:

- Not (x bar): inversion of boolean
- And (x && y, x . y): both are true
- Or: (x || y, x + y): at least one is true
- Xor: (x ⊕ y): exactly one is true

Bitwise Operations:

- Operates on bit by bit basis
- Bitwise Not (~x): inversion of each bit
- Bitwise And (&): and operation on bit by bit basis
    - Masking: letting some bits pass and not other bits
- Bitwise Or (|): or operation on bit by bit basis
- Bitwise Xor (^): xor operation on bit by bit basis
    - 0 ^ n = n
    - 1 ^ n = ~n
- Bitshift Left (<<): fill positions to the right with 0 (i.e., 1001 << 2 = 100100)
    - Each bitshift multiplies result by 2
- Bitshift Right (>>): fill position to the right with either 0 or 1 (i.e., 1001 >> 2 = 0010 or 1110)
    - Each bitshift integer divides result by 2
    - Arithmetic shift right fills using left most bit (for signed numbers)
    - Logical shift right fills using zeroes (for unsigned numbers)

Language of a CPU: variables, integers, floating points, arithmetic ops, assignment ops

- Restrictions: assign integers to variables and arithmetic on variables, two at a time

Core Components:

- Memory: hold the instructions as we operate on them
- Program Counter (PC): pointer to next statement
- Registers: holds the variables
- Arithmetic Logic Unit (ALU): performs arithmetic operations
- Instruction Register (IR): pointer to statement being currently executed

Functionality:

- Copy instruction from memory from where program counter points to the instruction register
- Execute instruction in instruction register (perhaps using the registers or the ALU)
- Update PC to next instruction (involves ALU) and repeat
- Need to involve operating system to talk to input/output devices
    - syscall: pauses program to interact with OS
    - system call code usually placed in $v0 (action)
    - argument usually placed in $a0 (value)

MIPS:

- RISC architecture: reduced instruction set computer
- file type is .asm
- instructions in 32 bits (4 bytes); 32 registers, each with 32 bits
- strings cannot be put in registers (common to put in memory)
- All arithmetic operations happen in registers
- Immediate Value: a number not stored in register
- addu: does not care about overflow
- multiplication: must use mult (two arguments) and mflo (destination register)
- not does not exist; NOT A = A NOR (NOT OR) 0

Pseudo-instruction:

- help us to use at high level "macro", but not actual "core" instruction in SPIM; not core to the CPU
- li: not actual instruction because there are not enough bits (32 for instruction, 32 for data)

Conditionals:

- set-less-than (slt): set some register to 1 if less than comparison of other registers hold true, 0 otherwise
- branch-equal-to (beq): compare if equal to, then go to next block (jump to new block) (else, continue)
- Pseudo-instructions: branch less than; branch greater than; branch less or equal, branch greater or equal

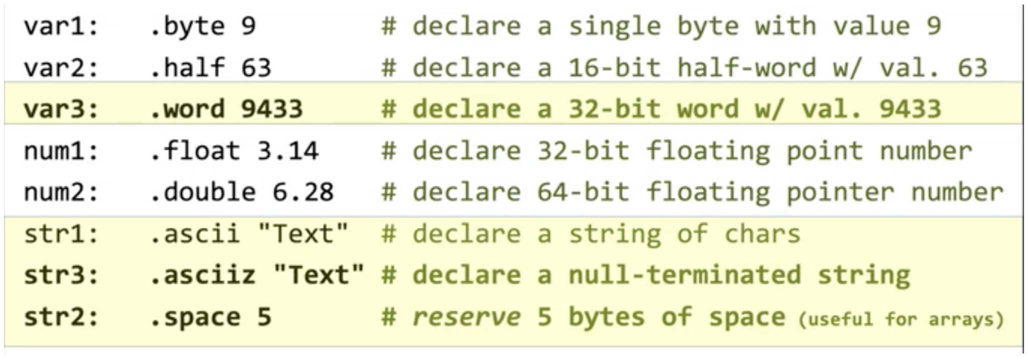
Loops:

- j command to jump back to same portion and repeat the loop
- condition at top with jump to exit loop

Accessing Memory: storage from RAM

- global (static) variables placed in memory not registers
- .data directive to declare variables, values, and names used
- Free Memory = stack + heap: allocated as program runs
- Initialized Data (constants) and Uninitialized Data (mutable global variables): allocated at program load
- must use in tangent with la to load address to register and then load word from address value
- Alignment restriction: addresses must start in multiples of 4
- use load word (lw) and store word (sw) to read and write to memory
- 1 word = 32 bits = 4 bytes = 8 hexadecimals
- Array: global variable with multiple elements

.data Declaration Types:



```
var1:   .byte 9        # declare a single byte with value 9
var2:   .half 63       # declare a 16-bit half-word w/ val. 63
var3:   .word 9433     # declare a 32-bit word w/ val. 9433
num1:   .float 3.14    # declare 32-bit floating point number
num2:   .double 6.28   # declare 64-bit floating pointer number
str1:   .ascii "Text"  # declare a string of chars
str3:   .asciiz "Text" # declare a null-terminated string
str2:   .space 5       # reserve 5 bytes of space (useful for arrays)
```

Memory Range: 32 bit addresses = 8 hexadecimals (always positive)

- Range: even though there's 32 bits of addresses, there's only a portion a programmer can use
    - only half is usable (256 Megabytes)
    - everything from 0x80000000 to 0xFFFFFFFF is not usable
- Big Endian: stores addresses from left to right (usually used by MIPS)
- Little Endian: stores addresses from right to left

Instruction Construction: MIPS to Assembly

- R Type Division (<op> <rd>, <rs>, <rt>): for operating two registers

			Bit Number
op code	6 bits	basic operation	(31 - 26)
rs code	5 bits	first register source operand	(25 - 21)
rt code	5 bits	second register source operand	(20 - 16)
rd code	5 bits	register destination operand	(15 - 11)
shamt code	5 bits	shift amount (for bit shift)	(10 - 6)
funct code	6 bits	function code	(5 - 0)

- 5 bits because stores 2^5 = 32 registers
- I Type Division (<op> <rt>, <rs>, immed): for immediate values

			Bit Number
op code	6 bits	basic operation	(31 - 26)
rs code	5 bits	first register source operand	(25 - 21)
rt code	5 bits	second register source operand (destination)	(20 - 16)
address code	16 bits	immediate constant or memory address	(15 - 0)

- address code is signed; limited to range of -2^{15} to $2^{15} - 1$ for immediate values
- for lw, the syntax is lw <rt>, <immed>(<rs>)
- J Type Division (<op> <label>): for jumping places in program

op code	6 bits	basic operation	(31 - 26)
address	26 bits		(25 - 0)

Functions

- need to execute blocks of code and pass arguments in and out
- jal (jump and link): jump to an address and store location of next instruction in register $ra
- jr (jump register): jump to addressed stored in a register, often $ra
- Convention: $a0 through $a3 are argument registers for passing function arguments
- Convention: $v0 and $v1 are return registers for passing return values
- For nested functions, must store function call locations in stack since only one $ra

Call Stack: store return addresses of various functions called

- lower address = top of stack
- $sp is a stack pointer to the top of the stack (smallest address)
  - any address smaller than $sp is garbage; any address greater is elements on the stack
- stack bottom: largest valid address
- stack limit: smallest valid address of a stack (how big the stack can become)
- Push register to stack: subtract 4 from $sp

Push:

```
addiu $sp, $sp, -4
sw $ra, 0($sp)
```

Pop:

```
lw $ra, 0($sp)
addiu $sp, $sp, 4
```

MIPS Calling Convention (CC):

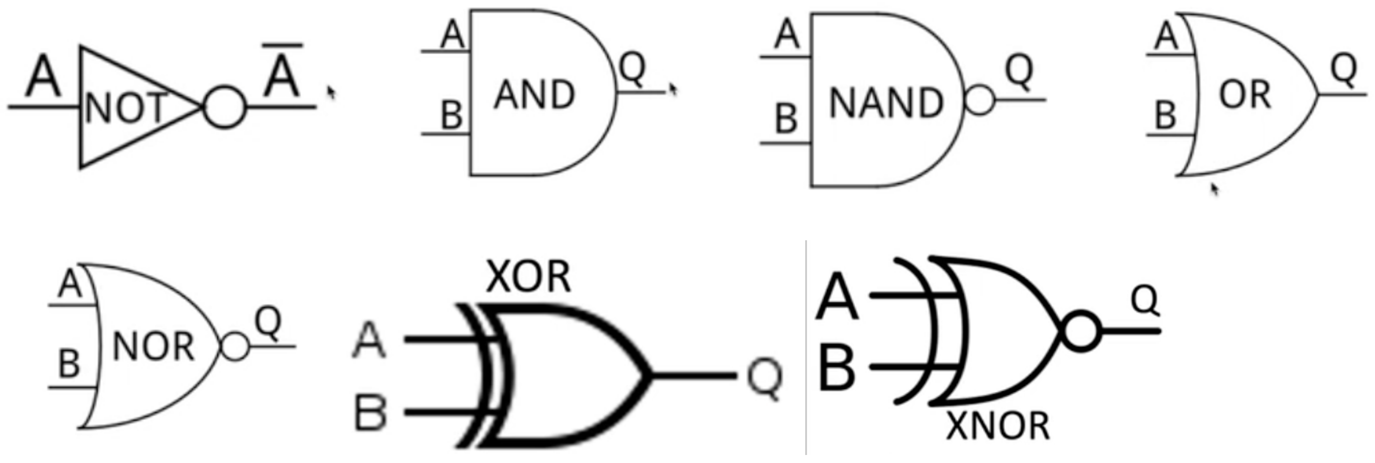
- Reasons to care:
    - make sure everyone follows same consistent and reliable methods
    - allow testing easier and standardized
- Calling Convention: assume anything not in $s registers is gone
- Protocol: rules for calling functions and return from functions
- Do not have inherent way of doing nested and recursive functions
- Assumptions:
    - Not utilize $fp and $gp
    - All values on stack 32-bits
    - Functions will take at most 4 arguments and return at most 2 arguments
- Preserved Registers: $s0 - $s7, $sp, $ra
    - push the s registers that will be used onto the stack
- Unpreserved Registers: $t0 - $t9, $a0 - $a3 (arguments), $v0 - $v1 (return values)

Recursive Functions: base and recursive case

- Same setup with nested functions, saving $ra and needed $s registers

Basic Building Blocks for Digital Logic:

- Logic Gates = Bitwise Operators: NOT, AND, OR, XOR, etc.
- Latency: time it takes for signals to traverse logic gates
- Half Adder: 1-bit adder without carry in bit



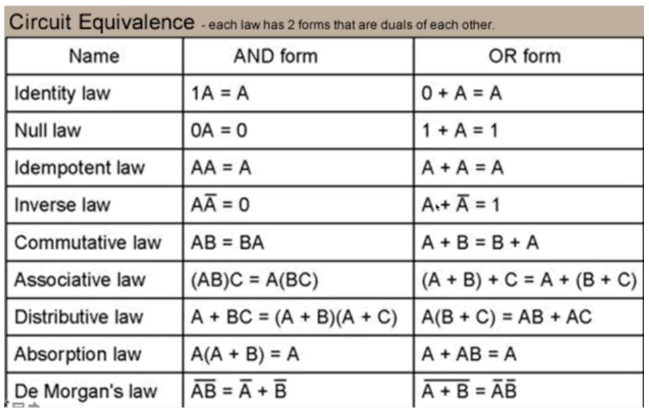
Truth Tables:

- # of entries = 2^n where n is the number of inputs

Logic Functions: output function seen as a logical combination of inputs

- Or = Logical Sum/Union (+)
- And = Logical Product/Disjunction (.)
- Minimization = Optimization for memory and space and reduces latency
- Simplified if there are less ORed expressions and/or fewer variables in the ANDed expressions => helps mitigate potential hardware issues with complex logic

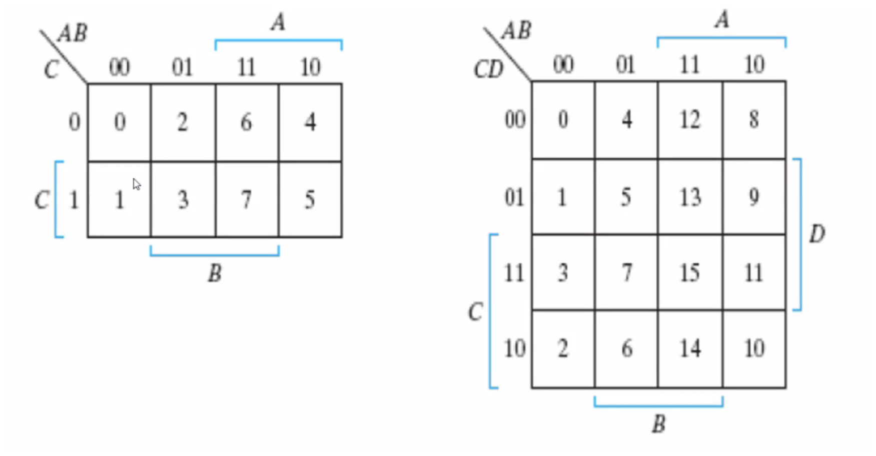
Logical Rules:



Circuit Equivalence - each law has 2 forms that are duals of each other.

Name	AND form	OR form
Identity law	$1A = A$	$0 + A = A$
Null law	$0A = 0$	$1 + A = 1$
Idempotent law	$AA = A$	$A + A = A$
Inverse law	$A\bar{A} = 0$	$A + \bar{A} = 1$
Commutative law	$AB = BA$	$A + B = B + A$
Associative law	$(AB)C = A(BC)$	$(A + B) + C = A + (B + C)$
Distributive law	$A + BC = (A + B)(A + C)$	$A(B + C) = AB + AC$
Absorption law	$A(A + B) = A$	$A + AB = A$
De Morgan's law	$\overline{AB} = \bar{A} + \bar{B}$	$\overline{A + B} = \bar{A}\bar{B}$

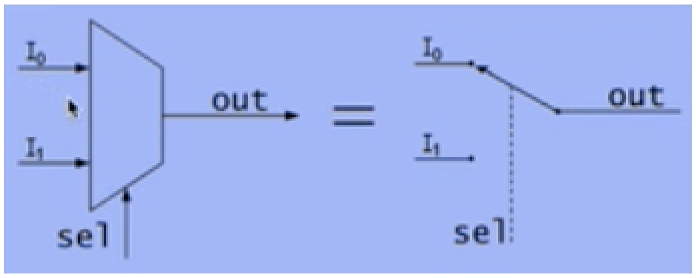
Karnaugh K-Maps:



- Group adjacent (not diagonal) cells containing "1"; cannot include elements "0"; wrapping around is good
- Groups must contain 2^n cells where n is an integer; can contain overlap to maximize group size
- Want to minimize number of groups and maximize number of groups
- Every cell containing a "1" must be in at least 1 group
- Don't care outputs are denote with an X; can exploit to simplify in K-maps
- Combinatorial Logic: combining multiple logics blocks together

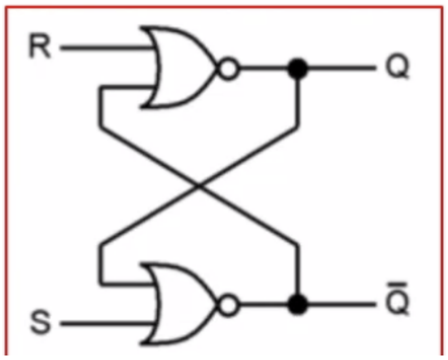
Multiplexer: selects input A or B to be the output given a third input the selector

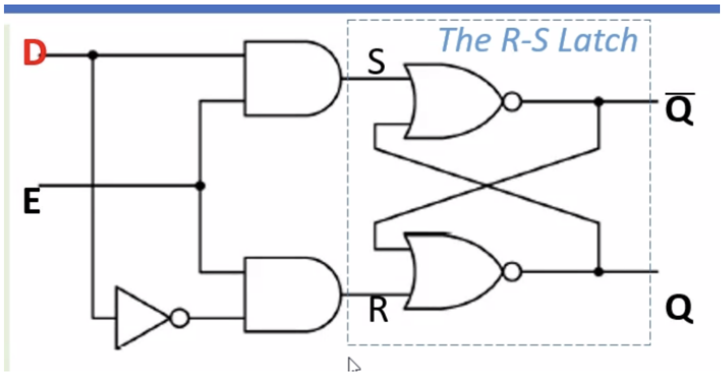
- if S == 0, return A, else if S == 1, return B
- Mux: N bits M inputs to 1 output
- Demux: 1 input to N outputs (1 to M)



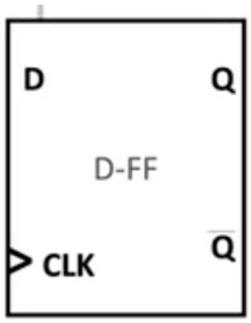
Set-Reset (SR) Latch: building block for memory

- set (R = 0, S = 1) = make output Q to 1; reset (R = 1, S = 0) = make Q to 0
- when (R = 0, S = 0), hold Q; when (R = 1, S = 1), Q is unable to be determined due to illegal operation
- feedback makes this latch unique to create memory in a digital design
- The Gated Data (D) Latch: forces S and R to be opposite when E (enabled) = 1; else both 0 and holds
- Each Gated D Latch serves as 1-bit of a register to store memory
- The Clocked D Latch: apply a synchronous clock on input E, where the clock alternates 1 to 0 periodically





D Flip Flop: capture D at rising edge (positive edge when 0 -> 1 of a clock)



Finite State Machines: abstract machine that is in one of finite number of states at any given time

- Defined by list of states, initial states, and conditions for transitions
- State: collection of outputs in a digital machine
- Machine: computational entity yielding a logical output given input conditions
- Moore Machine: output is function of previous input only
- Mealy Machine: output is function of present state and present input
- Unconditional Transition: does not depend on any input

.text	required starting line for instructions
#	comment
main:	label to start program
li <register>, <value>	load immediate numeric <value> in <register>
la <register>, <label>	load address of <label> in <register>
move <to register>, <from register>	copy value in <from register> to <to register>
add <register>, <value 1>, <value 2>	stores <value 1> + <value 2> in <register>
mult <register 1>, <register 2>	multiply value in <register 1> with value in <register 2>
mflo <register>	stores result of mult in <register>
sll <to register>, <from register>, <value>	bitshift <value> bits left
slt <set register>, <register 1>, <register 2>	<set register> = bool of <register 1> less than <register 2>
beq <register 1>, <register 2>, <label>	jump to <label> if <register 1> == <register 2>
bne <register 1>, <register 2>, <label>	jump to <label> if <register 1> != <register 2>
blt <register 1>, <register 2>, <label>	jump to <label> if <register 1> less than <register 2>
bgt <register 1>, <register 2>, <label>	jump to <label> if <register 1> greater than <register 2>
ble <register 1>, <register 2>, <label>	jump to <label> if <register 1> <= <register 2>
bge <register 1>, <register 2>, <label>	jump to <label> if <register 1> >= <register 2>
<label>: .asciiz <string>	declare global <string> referenced by <label>
lw <register> <N>(<address>)	dereference <address> register + <N>; store in <register>
sw <register> <N>(<address>)	store value in <register> back to <address> + N
$v0 = 1	print integer in $a0
$v0 = 4	print string in $a0
$v0 = 5	store input from std input in $v0
$v0 = 10	exit program
syscall	execute action in $v0
$zero	constant zero
$sp	stack pointer
$ra	return address
$t0 to $t9	temporary registers

$s0 to $s7	saved temporary registers
$a0 to $a3	argument registers
$v0 to $v1	function result registers