Free A+ CS2100 notes by: Calvin

**Lecture 1. Introduction**

- C is an imperative procedural language.

- needs to be compiled after each edit

- when using GCC, these happen: Preprocessing --> Compilation --> Assembler --> Linker

- GCC does it for us

**Lecture 2. C programming intro**

- Follows the edit, compile, execute cycle

General form:

preprocessor directives (eg #include <stdio.h>, #define MAX 10)

main function header

{

declaration of variables

executable statements (Input data (scanf or file input), Computation, Output results (printf or file))

}

Initially uninitialized variables might not be ZERO!! (Depends on what was there before)

data is stored in variables

- each variable has a name, data type, value, address

- C is strongly typed, each variable has to be declared with a data type

Common data types

- int: 4 bytes

- float: 4 bytes

- double: 8 bytes

- char: 1 byte

1. Preprocessor directives

- eg header files, macro expansions (for constant values)

- macro expansion just does a 1 to 1 replacement wherever it is, if there is data type conflict, it is resolved where the substitution occurs

2. IO

- scanf("%d", &age)

“age” refers to value in the variable age.

“&age” refers to (address of) the memory cell where the value of age is stored.

3. Compute

- Declarations statements: tell compiler what type of memory cells needed

-EG (int count) will allocate 4 bytes to count

- Executable statements: describe the processing on the memory cells

- Assignment statements

- Left side of ‘=’ is called lvalue

- lvalue must be assignable (cannot 32 = a, since 32 is not a variable)

- SIDE EFFECT: assignments will return the value of its right-hand side expression

- eg a = 12 will return 12, so you can b = a = 12 and b will be assigned 12

- Arithmetic operations

Precedence: those higher will be executed first

Primary expression operators

( ) [] . -> expr++ expr--

Left to right

Unary operators

\* & + - ! ~ ++expr --expr (typecast) sizeof

Right to left

Binary operators

\* / %

+ -

< > <= >=

== !=

&&

||

Left to right

Ternary operator

? :

Right to left

Assignment operators

= += -= \*= /= %=

Right to left

Truth values

- In C, 0 represents false

- ANY OTHER value is used to represent true (can be 1 or any other number)

Short-circuit evaluation

- expr1 || expr2: If expr1 is true, skip evaluating expr2 and return true

immediately, as the result will always be true.

- expr1 && expr2: If expr1 is false, skip evaluating expr2 and return

false immediately, as the result will always be false.

**Lecture 3. Data representation & Number system**

Basics:

- N bits can represent up to 2^N values

- To represent M values, ceiling (log2 M) bits required

- 0x --> hexadecimal

- 0b --> binary

- 0 --> octal

Binary

- From decimal to binary whole numbers, repeatedly divide by 2 and get the remainder, read from bottom to top

- for fractions, repeatedly multiply by 2, the bit is 1 if there is a carryover (ie 0.75 \* 2 = 1.5 so theres a carry of 1). Read from top to bottom

to convert from bin to octal: partition in grps of 3 starting from the back and get the octal representation of each grps

(reversed for octal to bin)

to convert from bin to hexa: same as octal but in grps of 4

Signed numbers

1. Sign-and-Magnitude

2. 1s Complement

3. 2s Complement

4. Excess representation

1. Sign-and-Magnitude

- first bit is for parity (0 == +, 1 == -)

- the remaining bits is the number in binary padded with zeros

§ Largest value: 01111111 = +127

§ Smallest value: 11111111 = -127

§ Zeros: 00000000 = +0 (2 types of zeros)

10000000 = -0

§ Range (for 8-bit): -127 to +127

- to negate, just invert the sign bit at the front

2. 1s Complement

- for positive numbers, its negated value is -x = 2^n – x – 1

- eg to negate 00001100, it is -00001100 in binary but 11110011 in 1s-complement

- just invert the bits

§ Largest value: 01111111 = +127

§ Smallest value: 10000000 = -127

§ Zeros: 00000000 = +0 (Had 2 types of zeros)

11111111 = -0

§ Range (for 8 bits): -127 to +127

§ Range (for n bits): -(2^(n-1) – 1) to 2^(n-1) – 1

Adding and subtracting:

1. Perform binary addition on the two numbers.

2. If there is a carry out of the MSB, add 1 to the result.

3. Check for overflow. Overflow occurs if result is opposite sign of A and B.

For subtraction, take 1s-complement of B and add it to A

3. 2s-complement

- For positive numbers, its negated value in 2s is -x = 2^n - x

- eg 00001100 negated will be 11110100 in 2s-complement

- trick: just invert everything to the left of the right-most 1

- OR: just find 1s complement then add 1

§ Largest value: 01111111 = +127

§ Smallest value: 10000000 = -128

§ Zero: 00000000 = +0 (ONLY 1 ZERO, there is no negative 0)

§ Range (for 8 bits): -128 to +127

§ Range (for n bits): -2^(n-1) to 2^(n-1) – 1

adding and subtracting:

1. Perform binary addition on the two numbers.

2. Ignore the carry out of the MSB.

3. Check for overflow. Overflow occurs if the ‘carry in’ and ‘carry out’ of the MSB are different, or if result is

opposite sign of A and B.

For subtraction, take the 2s complement of B, then add it to A

4. Excess representation

- Allows range of value to be distributed EVENLY between positive and negative values

- Excess-4 means that in excess-4, 101 is 1, to get the actual binary representation of 101 just add 4+ 1 which is 5.

- to convert binary to excess-4, 111 is 7 in binary but 7-4 =3 in excess-4

Fixed-point representation

- used to represent real numbers

- number of bits allocated for the whole number part and fractional part are fixed

- has range limits as it is not flexible

Floating-point representation

- allows for very large or very small numbers

- Single precision format (32-bits): 1-bit sign, 8-bit exponent with excess-127, 23-bit mantissa

- need to convert to scientific notation first ie to 1.101 x 2^2

Sign bit: 0 for +ve, 1 for -ve

exponent: add 127 to the original exponent

mantissa: since the original number is 1.101 x 2^2, we can just ignore the leading 1, so we just store 101

Graphical user interface

Description automatically generated with medium confidence

**Lecture 4. Pointers and Functions**

- Pointers allow direct manipulation of memory contents

- eg

int a = 123

printing a will give 123, while printing &a will print the address of a

- C has bitwise operators to allow efficient bitwise operations

Pointers:

- pointers are variables that contains the address of another variable

- eg a\_ptr is a pointer that holds the address of variable a

- To initialise a pointer: type \*pointer\_name

- eg to declare a pointer to the address of a, we can use int \*a\_ptr = &a;

- Accessing the variable through pointers

- since a\_ptr already points to a, it is identical to a, printing a and \*a\_ptr will give 123

- \*a\_ptr is the dereferencing operator (to access the address's content)

- for simplicity, int \*b can be read as "b is a pointer to some integer"

double a;

double \*b;

b = &a;

is the same as:

double a;

double \*b = &a;

- uses:

- pass address of variables so tht the function can change the values

- pass address of array head so that the func can access the whole array

Functions:

- function prototype is needed so that the compiler knows the output type

- without a prototype, C assumes that it is int

- pass-by-value:

- in C, the input parameters are passed by value, meaning the original value is unchanged

- the soln is to use pointers

- we can pass in pointers to the function and work with the pointers

- eg swap(&a, &b) instead of swap(a, b) and the swap func takes in (int \*a, int \*b)

explanation:

- Formal parameters are local to the function they are declared in.

- Variables declared within the function are also local to the function.

- Local parameters and variables are only accessible in the function they are declared – scope rule.

- When a function is called, an activation record is created in the call stack, and memory is allocated for the local parameters and variables of the function.

- Once the function is done, the activation record is removed, and memory allocated for the local parameters and variables is released.

- Hence, local parameters and variables of a function exist in memory only during the execution of the function. They are called automatic variables.

**Lecture 5. Arrays, Strings and structs**

Arrays:

- homogeneous collection of data

- declare: int arr[10] need to declare the size!

- the name arr refers to the address of the first elems, it is a fixed pointer to the first elem

- equilivant to &arr[0]

- since it is just a pointer, we can do

int sumArray(int \*arr, int size) instead of

int sumArray(int arr[], int size)

- Since it is already a pointer, we can directly modify an array

- array elems occupy contiguous mem locations, are accessed thru indexing

- initially the elems in an array are undetermined, cannot assume it is 0

Strings:

- is just an array of char with a null character '\0'

- is used by many string functions to know when the string terminates, many functions will continue to run and access illegal parts of memory without it

- there is a string library <string.h> to manipulate strings

- Initialising strings

§ Two ways:

• char fruit\_name[] = "apple"; (\0 is automatically added)

• char fruit\_name[] = {'a','p','p','l','e','\0'};

-Strings I/O:

§ Read string from stdin (keyboard)

fgets(str, size, stdin) // reads size – 1 char, or until newline

- however might need to replace the new line char with \0

scanf("%s", str); // reads until white space

§ Print string to stdout (monitor)

puts(str); // terminates with newline

printf("%s\n", str);

- Common String funcs:

- strlen, strcmp, strncmp, strcpy, strncpy

Structs:

- allow grouping of heterogeneous members (of diff types incl other grps)

examples:

typedef struct {

int acctNum;

float balance;

} account\_t;

typedef struct {

int length, width, height;

} box\_t;

NEED to include the name of the struct

- Similar to normal var, when a struct variable is passed to a func, a separate copy is made (original will not be modified)

- if you want to modify it, need to use pointers again

- eg if we want to change the name of a player, we use use (\*player\_ptr).name which is equivalent to player\_ptr->name

**Lecture 7. MIPS (assembly language)**

ISA:

- abstraction on the interface between hardware and low-level software

- does the translation to instruction set

- tells you how to perform an add or subtract etc

- abstracts away the hardware and lets you talk abt functions

- program compiled for the same ISA, can be run on any implementation of ISA

machine code:

- instructions in binary, but written in hexa for simplicity

Assembly lang:

- symbolic ver of machine code

- human readable, but need an Assembler to translate from assembly to machine code

components in computer

- processor and memory

- processor does the computation

- has ALU (arithmetic logic unit to calc)

- mem stores the code(instructions) and data

- connected by a Bus (3 grps of buses)

- address bus (cpu to mem) -> sends index no to mem and either put a piece of data at tt index or takes a piece of data from thr and gives to processor

- data bus (bi-directional)

-control bus (cpu to mem) -> tells mem to read or write

- processor is 240x faster than mem

Registers:

- cpu has registers to temp store values so that operations can be performed on registers (no need

access mem, and is much faster)

- only 16 to 32 registers (MIPS has 32)

- registers has no data type. just stores 32bit

- the data type is based on the instruction given

- EXCEPT Load and Store func which load from mem to register, and from register to mem respectively

- only 2 instructions that access mem directly (except from intel processors)

- will map the variables in memory, like saying r1 = res

- after the program, need to use Store to send the data back to the mem

- referred by a number eg $0, $1 ...

- diff registers have diff specific purposes

- 0, 1, 28-31 shld not be touched for reasons

Control flow

- need instructors to change the control flow based on conditions (loops and if-else)

- location specified in the ISA

MIPS assembly language

- has very simple instructions, each line has at most 1 instruction

- General syntax

- 3 operands: 2 sources and 1 destination

- 1 operation

arithmetic operations

- most MIPS math operations are register-to-register

- eg:

a = b + c - d;

becomes (need be broken down into 2 operations)

add $t0 , $s1, $s2 #tmp = b + c ($t0 to $t7 are temp registers)

sub $s0 , $t0, $s3 # a = tmp - d

- has addi instruction to add a constant

- the constant ranges from [-2^15 to 2^15 -1) (stored in 16bit 2s complement)

- large constants are loaded 16 bits at a time using lui and ori to the register

- Register zero

- $zero always has the value zero

- used for assignment operations like f = g, which would translate to

--> add &s0, $s1, $zero

OR move $s0, $s1 (syntactic sugar)

Logical Operations

- view registers as 32 raw bits

- AND, OR, NOR, XOR

-same as in 1231

- AND operation can be used for masking (ie extracting certain bits from a word)

- put 1 in the positions you want to extract, 0 everywhere else

- OR operator can be used to force certain bits to 1

- NOR instruction

- can be used as a NOT by using $zero as operand

- XOR instruction

- can also be used as NOT by xor all 1s

- Shifting

- sll and srl (shift left/right logical)

- can only shift a max of 32 bits left or right

- shifting left n bits will multiply by 2^n, reverse for shift right

**Lecture 8. MIPS pt 2**

Memory instructions

- each address contains 8bit (1 byte)

- has 2^32 -1 addresses

- MIPS is a load-store register architecture

- like an array, can use the index number to get the content

- we can access a single byte or a single word (4 bytes/ 32 bits in MIPS)

- Words are the most common unit of transfer between processor and mem

- is the register size, integer size and instruction size

Word alignment

- words are aligned in mem if they begin at a byte address that is a multiple of the num of bytes in a word, eg

if its from byte 0 to 3 it is aligned

- if you try to load an unaligned word, it’ll crash

- MIPS has 2^30 mem words (ensure that they are aligned)

1. Load word eg (lw $t0, 4($s0)

- specify mem address

-stored in 2 parts (source register + displacement)

- mem word at the address is loaded into $t0

2. Store word eg (sw $t0, 12($s0)

- same as load word, specify the mem address

- mem address = $s0 + 12

- content of $t0 is stored into the word at that mem address

3. Others

- MIPS can also load or store a single byte with lb and sb

- offset no need be multiple of 4 (no need be aligned)

- MIPS can store half words (16 bits instead of 32) and unaligned words (for unaligned words)

Arrays in MIPS

- in general, to access A[i], the address is $s3 + 4i (becos of the offset)

Control flow instructions

- can alter the control flow

- can change the next instruction to be executed

1. Conditional branch

- beq (branch if equal)

- beq $r1, $r2, L1 --> if (a == b) goto L1

- bne (branch if not equal)

- opposite of beq

2. jump instruction

- j (jump)

- is unconditional, will always just jump to L1

3. If statements

- uses a mixture of beq and bne,

- usually, we have to invert the condition for shorter code

4. Loops

- uses labels and goto statements to loop

5. inequalities

- we use slt and slti (set on less than)

- sets a variable to be a number, if $s1 is < $s2

**Lecture 9: MIPS 3**

MIPS encoding

- all instructions are fixed-length at 32 bits

- makes it easier to convert to binary (consistent)

- 3 instruction types

- R-format (register format eg add, sub, srl, sll)

- I-format (Immediate format eg addi, andi, lw, sw, beq, bne)

- J-formal (jump instruction)

1. R format (32 bits total)

- register numbers are 5 bits each (since they go from 0-31)

- rs (source 1), rt (source 2) and rd (destination) registers

- shamt (shift amount) is also 5 bits (since can shift from 0 - 31 only)

- 0 for all non-shift instructions

- opcode and funct are 6 bits each

- works tgt to specify the instruction

- opcode in Rformat is always 0

2. I format

- normally in Rformat, the shamt field can only have 5 bits, but we need 16 bits

- since we only using 2 registers, and we dun need the funct field (opcode handles everything), and the shift instructions are alr immediately

- We combine rd, shamt and funct to have the immediate instruction

- the rest are the same

- immediate:

- singed integer (2^-15 to 2^15 -1) in 2s complement

- large enough to handle lw or sw and all the addi etc

- PC-relative addressing (programme counter) -> for bne and beq

- allows us to jump to different memory address using only a 16-bit number

- the 16-bit immediate is used in addition to the programme counter (tracks the current

address)

- immediate keeps track of the number of instructions to skip forward or back to

- if branch if not taken, just PC = PC + 4 (PC + 4 is the address of the next instruction)

- if branch is taken PC = (PC + 4) + (immediate \* 4)

- immediate is number of instructions, since each instruction is 4 bytes, we need \*4 to get the correct address

3. J format

- for general jumps where you can jump anywhere in memory

- uses first 6 bits for opcode, then 26 bits for the address to jump to

- due to optimization (we use instructions rather than address; we can have 28 bits)

- to get the full 32-bit address to jump to (all them all tgt):

1. take the most significant 4 bits from PC + 4

2. take the 26 bits stated in the instruction

3 default 00 bits as it is word aligned

**Lecture 10: Instruction Set Architecture**

- Mips is a type of ISA

- CISC vs RISC

1. Data storage

- how to store operands and result etc (eg. registers)

- eg stack architecture, accumulator, general-purpose register, memory-memory

2. memory and addressing mode

- Memory Address Register --> stores memory address to access

- Memory Data Register --> stores data to be written or to be read

- endianness (how to store a word in memory, eg MSB in largest address or reverse)

- big endian --> MSB stored in lowest address

- little endian --> LSB stored in lowest address

addressing mode:

- ways to specify and operand

- in MIPS there’s 3 types: register, immediate, displacement

3. operations

- types

- data movement (from memory to register)

- arithmetic

- control flow

4. Instruction formats

- instruction length

-> RISC length of 1 word (32 bits or 64 bits)

-> CISC variable length instructions

- instruction fields -> the type and size of operands

- opcode and operands

5. encoding the instruction set

- instruction encoding -> how the instructions are represented in binary?

- encoding choices

1. variable

2. fixed

3. hybrid

fixed length instructions

- expanding opcode scheme (opcode has diff length for diff instructions)

- allows for unused bits to be used as opcode

- allows for (2^6 -1) \* 2^5 + 1 = 2017 instructions

Formula for max num of inst:

2N0 – 2N0 – N1 – 2N0 – N2 - … + k - 1 (N0 is largest opcode bits, N1 is 2nd largest…, k is total inst class)

Formula for min num of inst:

2N0 + 2N1 – N0 + 2N2 – N1 + … - k + 1 (N0 is smallest opcode bits, N0 is 2nd smallest…, k is total inst class)

**Lecture 11: Datapath**

- How the encoding is used to do operations

- is one of 2 major components (other is control)

- takes in data from operands, process it, and writes data back

5 stages of data path:

1. Fetch

- get instruction from memory,

- address is in PC register (points to instruction to be fetched)

2. Decode

- understand the instruction (e.g. add, slt etc)

3. Operand fetch

- gets the operands (contents of the registers)

combined with Decode stage (as decode is easy for MIPS)

4. Execute

- performs the operation

Split into ALU and Memory access stage (as sometimes it is not needed)

5. Register Write

- writes the result of operation to register

Build a data path from each stage

1. Fetch stage

- Need to access PC (to get the instruction)

- Need to increment by 4 (as instructions are 32 bits long)

- outputs to decode stage

Instruction Memory

- stores info (instructions that has been encoded)

- takes the address of the instructions, and passes the content to decode stage

- timed with a clock

- PC also uses a clock

- will only update when there’s a rising clock edge, when not updating, the counter is stable, can be

used to extract info from memory

Adder

- Takes in 2 32-bit number, and outputs another 32 bit number

2. Decode stage

- need gather data from instructions fields

- read data from all necessary registers (can be 2(add), 1(addi) or 0(j)

Register file

- contains all the registers

- read register is the input register to read from

- read data is the info from the register

- has a control signal RegWrite to indicate if we need write (is just binary 0 or 1)

For I format:

- The order of rt is mixed up, rt shld go to write register

- need a Multiplexer to choose which input is the write register

- for i-format, choose rt,

- for r-format, choose rd to go to write data

if i have n inputs, will have log2(n) lines in the control (encoded in binary)

- Need ANOTHER multiplexer

- for rformat and iformat, the 2nd operand is different

- controlled by ALUsrc (0 means come from read data 2, 1 means come from the instruction)

- if the immediate field is chosen, need sign extend as ALU works in 32 bits inputs

For branching

- need to do more computation in ALU

3. ALU stage

- AKA execution stage

- performs all the work (arithmetic, shifting, logical, memory: address calc, branch operation etc)

ALU

- has 2 outputs, result, and isZero

- ALUcontrol sends the control for the operation

For non-branch instructions:

- straight forward

For branch instructions:

- we need perform 2 calculations

- branch outcome

- 1-bit isZero signal

- branch target address

- Comes from another adder connected to PC and immediate field

- that is why we need calc from PC+4 (due to the clock cycle)

- The MUX there is to send both posb outcomes, the target branch if the condition is true, or

PC+4 if it is not true

- We need change the ALUsrc to be set to 0 as that it reads from register instead of immediate

- handled by the control

4. Memory access stage

- load and store instructions

- needs memory address & data to be written to data memory

- only lw and sw will trigger this stage

- will output results ONLY for lw

Data memory

- stores data values

- has 2 inputs (mem address and data to be written)

- has 2 controls -> read and write control, ONLY 1 can be asserted at a time

Load Word instruction

- I format instruction, immediate is sent to write register

Store word

- similar as lw, except we need a connection from RD2 to write data in data memory

non memory instructions

- since we dun wan to access memory, we need another MUX to bypass the data memory, instead

we just get the ALU result

5 Register write stage

- write the results of computation into a register

- need dest register number and result from computation

- only stores, branch and jump dun need to write to register

- Just need route the result from mem stage to the Register File

**Lecture 12: Control**

- control signal is generated based on instruction to be executed

- TIP: 000000 is r-format, 000010 is j-format, everything else is i-format

- for r-format, we need funct field to generate the control signals

RegDst

- Choses between rd and rt for the dest

RegWrite

- Choses if we need to write from WD to WR

ALUSrc

- chooses what goes into ALU

- first input is content of rs register

- 2nd input comes from 2nd register or sign extend

- ALUSrc choses the 2nd input

MemRead

- chooses to read the memory using Address

MemWrite

- choses to write data to address

MemToReg:

- chooses betw the data read from mem or result from ALU to be returned to register file

- FOR SOME REASON, THIS MUX IS INVERTED, 1 IS THE TOP INPUT, 0 IS BOTTOM

PCSrc

- choose between PC + 4 or PC + 4 + 4\*imm (for branch instructions)

- is determined by isZero from ALU + branch instruction

ALUControl

- need the opcode + funct fields

- The actl ALU is 32 bits, with each 1-bit ALU slice feeding in to the next (Cout -> Cin)

- Ainvert and Binvert inverts in input bits if needed

- Operation controls the MUX

- 0 = AND

- 1 = OR

= 2 = adder

- Binvert is used for subtract as A + B' = A + (-B) in 2s complement

- We can use an intermediate control called ALUop

- generated from the OPcode

- used mainly for lw, sw and beq

- for R-type, it is all 10 for the ALUop

- ALUcontrol is 4 bits,

- bit 3 is always 0

- bit 2 is 1 if the LSM of ALUop is 1 (beq) or F1 is 1 (sub and slt)

- bit 1 is a 0 if MSB is 1 AND F2 is 1 (and and or)

- bit 0 is 1 if (MSB is 1 and F0 is 1) OR (MSB is 1 and F3 is 1)

Combinational circuit implementation

- we use inverters and an AND gate to check for inputs

**Graphical user interface, application, table, Excel

Description automatically generated**

Table

Description automatically generated

Table

Description automatically generated

**Lecture 13: Boolean Algebra**

Digital circuits

- square waves

- types of circuit

- combinational

- sequential

Boolean algebra

- Conjunction (AND) (.)

- disjunction (OR) (+)

- negation (NOT) (')

Order of operators

- Not > And > OR

eg. A . B + C = (A . B) + C

Laws Of boolean algebra

- same as in cs1231s

Duality

- by swapping . and +, 1 and 0, we can easily find the dual of the other equation

eg a+(b.c) = (a+b).(a+c) is equilivant to a.(b+c)+(a.c) --> they are duals of each other

Theorems:

complement functions:

- as long as the func outputs the complement of the original fucn

it is a complement

- can be gotten by negating the original func

Standard forms

Every boolean expr can be expressed in SOP or POS form

- literals -> just a single variable or its complement

eg. x or y'

- product term -> a single literal or logical product (AND) of several literals

eg x, A'.B, x.y.z

- sum term -> single literal or logical sum (OR) of several literals

eg. x, x+y+z', A+B

- Sum of Products (SOP) -> product term or a logical sum (OR) of several product terms

eg x, x + y.z', A.B + A'.B'

- Product of Sums (POS) -> a sum term or a logical product (AND) of several sum terms

eg. x, x.(y+z'),

minterms, maxterms

minterm

- product term that contains n literal from all the variables

eg for variables x and y, as long as x.y or x'.y ...

maxterm

- sum term that contains n literals from all the variables

eg x+y, x+y' ..

in general, with n variables, we have 2^n min and max terms

- cos need include all the variables

each minterm is the Complement of the maxterm and vice versa

eg m2' = M2

Canonical forms

- unique form of representations

- sum-of-minterms = Canonical SOP

- given a truth table to find SoM

-> find the 1 values and sum them up

- product of maxterms - Canonical POS

- given a truth table to find PoM

-> find the 0 values and multiply them up

- can easily convert between SoM and PoM easily

-> just take the 'complement"

**Lecture 14: Logic circuits**

Types:

- Inverters

- AND gate

- OR gate

- NAND gate -> AND gate with inverter after

- NOR gate -> OR gate with inverter after

- XOR gate

- XNOR gate -> XOR gate with inverter after

Universal gates

- AND/OR/NOT are the complete set of logic

- other gates are used for usefulness, economical, self-sufficient

- NAND is self-sufficient, can implement NOT/AND/OR with just a NAND

- NOR is also self-sufficient

**Lecture 15: Simplification**

Graphical user interface, application, table

Description automatically generatedTable

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Half adder

- used to add 2 literals, split into 2 outputs (sum and carry)

- C = X . Y

- S = X.'Y + X.Y' = X XOR Y

Grey code

- DIFF from binary

- as long as from each value ONLY one bit changes, is a grey code

- MUST not have dupes

- need not start from 0, as long as cycles thru all the values

K-map

- obtain simplified SOP expr

- works up to 6 levels, but it gets funky, so best is 5 and lower

- aim to find grps in the k-map, the bigger the grp, the more variables we can remove

Diagram

Description automatically generated

Prime Implicant

- the largest grping possible

Essential prime implicant

- includes at least one minterm that is not covered by any other prime implicant (means covers at least one space tt no other PI covers)

**Lecture 17: Combinational Circuits**

Analysing a given combi circuit

1. label the inputs and outputs

2. Obtain the func of the intermediate pts and outputs

2 approaches to designing combi circuits

- gate-level and block-level designs

1. Gate level (SSI) design

- draw truth table -> k map -> form SOP -> logic circuit

2. Block level design

- need more creativity

Magnitude comparator

- compares 2 unsigned values (0 or more)

- normally with gatelevel design, need 2^(2n) rows in truth table cos each value need n bits

- only if the bits are equal, then we need move on to the next bit

- works using effectively XNOR gates (only outputs 1 if both A and B are the same)

- the rest works using 4 ORs to compare the 4 cases,. where each one waterfalls down

Circuit delays

- the time taken for all the inputs to arrive is max(of all the inputs) + t(delay of this circuit itself)

Some tips:

- FA can be used to add up 3 bits tgt,

- 2-bit adder also can

- 4 bit parallel adder can be used to implement (ABCD + 1)/2

Chart, diagram

Description automatically generatedDiagram

Description automatically generated

**Lecture 18: MSI Components**

1. Decoder

- takes in n inputs and outputs 2^n outputs

- some of them have an enable signal which is just a 0 if you want the decoder to be working

- some enables are 1-enabled (zero-enable is more common)

- enabled can be identified by a complement sign usually

- how to build decoders?

- use smaller decoders and with enables ( + a NOT gate) to build up larger decoders

- some of them also have active high and active low outputs (negated outputs)

- negated output --> can think of it as reversing the output

Diagram, schematic

Description automatically generated

2. Encoder

- opposite of decoder

- only if there is one 1 is a valid code, else is all dont cares

Priority encoders

- has priority, if there is an invalid input (other than all 0s), will still give an output

- just fill in all the dun cares

3. Multiplexer & Demultiplexers

- shared communication line between multiple devices, only one source and dest can use the line at once

Demultiplexer

- similar to a decoder with enable except the enable is replaced with a data line

Diagram, schematic

Description automatically generated

Multiplexer

- is a data selector, can have multiple inputs, but only 1 output

- can be made from decoders + some AND gates

- larger multiplexers can be built from smaller multiplexers

- need check the inputs are CBA or ABC (C is the MSB)

Implementing functions

- if the multiplexer is the correct size, just input the minterms that you wan

e.g. if we need F(A,B,C) = m(1,3,5,6) then just set the inputs of 1,3,5,6 to 1, the rest to 0

- if the multiplexer is too smaller, need break up the rows into block of 2, then see the wanted F value,

if the F is both 1s, then MUX in is just 1, same for 0

Diagram

Description automatically generated

**Lecture 19: Sequential Logic**

Memory element:

- can rmb values based on inputs, outputs based on stored value

- some have a clock which is square wave with positive and negative edges

- 2 types of triggering/activation

1. pulse triggered -> latches (on the peaks)

2. edge triggered -> flip-flops (on the going up and down portions, needs clock)

SR Latch Active high(NOR latch -> made of 2 NOR gates)

- 2 inputs S and R (SET (1) and RESET (0)

- 2 outputs Q and Q'

- Q can be High (SET) or Low (RESET)

- S = 1, R = 0 will set Q to 1

- S = 0, R = 1 will set Q to 0

- S = 0, R = 0 will set Q to be unchanged from prev state

- S = 1, R = 1 is invalid (inf loop)

Variations:

Active-low SR latch (using NAND gates)

- its completely opposite of the active-high one

Gated SR latch

- SR latch + enable input + 2 NAND gates

Gated D latch

- takes in D input instead of S and R

- 1 is SET, 0 is RESET

- D is split into 2, one of it is negated, ensures that the S and R are complements

- Removes the no change command

- removes the potential for invalid inputs

Flip-Flop

- synced version of latches, easier to coordinate between devices

- uses the rising and falling edges of the clock signal

- has positive (activated on rising edge) and negative (activated on falling edge) edge-triggerd flip flops

- functionally identical as the pulse are fast

- Has SR flip flops, D flip flops (used commonly in memory)

- sink: once it enters this state, it never moves out of that state

- unused state: no input will go to that state

JK Flip Flop

- No invalid state

- J = 1, K = 0 will set Q to 1

- J = 0, K = 1 will set Q to 0

- J = 0, K = 0 will set Q to be unchanged from prev state

- J = 1, K = 1 will toggle Q

-Q(t+1) = JQ' + K'Q

T Flip Flop

- 1 input version of JK flipflop

- If T is 0, Q no change

- If T is 1, Q toggles

-Q(t+1) = TQ' + T'Q

Async inputs

- used to init the initial state of the flip flop using PRE(1) and CLR(0)

-immediately does it, FORCES it to switch states regardless of clock cycle

- for active high (ACTIVE LOW is opposite) ,

PRE=1 -> Q = 1,

CLR=1 -> Q = 0,

both PRE and CLR = 1, Flip flop normal function

Analysis:

- From the circuit, generate a characteristic table

- 1 flip flop = 2 outputs, 2 ff = 4 outputs, 3 ff = 8 outputs

- From the circuit, find the input functions, then fill in the flip flop inputs, then fill in the next state

Design:

- From a state diagram, generate the state table

- fill in the flipflop inputs then draw the circuit (can have dont cares)

- for unused states, the flip flops are all dun caresDiagram, engineering drawing

Description automatically generated

Table

Description automatically generated

Table

Description automatically generated with medium confidence

Memory:

- how to use flip flops to build larger memory blocks

- Processor contains 2 impt registers

- MAR (mem addr reg) ->

- k bits can access 2^k addresses

- MDR (mem data reg) <->

- n bits is the size of the data chunks (word size)

single mem cell

- 1 bit select (enable)

- 1 bit input

- 1 bit output

- 1 bit read/write

- read will output Q

- write will change Q to input

4x3 RAM (4 word, each word 3 bits)

- since 4 words, need 2 bits to address it

- this time, we write or read from 1 row of BC

TO build bigger chips

- we use smaller clips w/ decodes to direct to a smaller mem chip

**Lecture 20: Pipelining**

- idea: have tasks sequenced in a way to make it more efficient

- sever instructions happening at the same time but diff stage of the process

- helps the throughput of the workload

- need consider:

- dependencies

- diff length inst times

MIPS pipelining:

- the MIPS data path can be split into 5 stages

1. IF instruction fetch

2. ID instruction decode

3 EX execute

4. MEM memory access

5. WB write back

- needs pipeline registers between each stage

- 4 pipeline regs

- used to carry data between each stage

- WRITE reg is passed thru the whole pipeline to ensure that the reg

to be written to is correct

1. IF/ID pipeline reg

- stores

- PC+4 and

- instruction read from InstructionMemory

- supplies

- reg numbers for the 2 input regs and

- 16-bit offset for sign-extend

2. ID/EX

- stores and supplies

-data values read from reg files

- 32 bit imm value

- PC + 4

3. EX/MEM

- stores

- (PC + 4) + imm\*4

- ALU result

- isZero?

- Data read 2 from reg file

4. MEM/WB

- stores

- ALU result

- mem read data

- needs to pipeline the control

- certain control signals are only used in certain pipelining stages

- ID/EX stage stores all 9 control signals (4 are used in EX stage)

- EX/MEM only has 5 of the control signals (3 are used in MEM stage)

- MEM/WB only has 2 of the control signals

- comparison of performance

single cycle

- take the max time instruction, tt is the standard time for each cycle

multi cycle

- take each cycle time \* average Cycle per instruction \* instructions

pipeline

- find max time of all the stages stage + overhead

- number of cycles needed for I instructions --> I + N - 1 (need N-1 cycles to fill in the pipeline) (N is the number of stages)

- time = (I + N - 1) \* (max time + overhead)

- optimising pipeline

- each stage is same time taken

- no overhead

- I >> N (the number of cycles is just I)

total: N times speed up

**Lecture 21: Pipelining Hazards**

Structural hazards

- need to access the same component at the same time

1. clash in loading and storing to mem

-soln: use 2 diff mem hardware, one for data and one for instruction (no clash).

2. clash in reg

- soln: split into 2 halves as regs are super fast (first half for writing, 2nd half for reading)

Data dependencies

- if a later inst needs data from earlier instruction, but both are run concurrently

- Read After Write (RAW)

- when access old mem before it has been written to

- Soln: use forwarding (bypassing)

- just pass the result from the computation directly to next inst instead of waiting to store then read from mem

- HOWEVER, Load word is jank af and cannot use forwarding as the result is only computed in stage CC4 ( memory), not in sync with the other instruction

- need to delay one cycle

if no forwarding:

- 2 cycles of delay if inst is immediately after the writing inst

- 1 cycle of delay if inst is 1 inst after writing inst

with forwarding:

- NO cycle of delay for most inst

- 1 cycle of delay if inst is 1 inst after LW

eg. sub $2, $1, $3

and $12, $2, $5 #2 cycle of delay

or $13, $2, $6 #1 cycle of delay

* Hazards

1. Read-After-Write: 2 stalls without forwarding, no stalls with forwarding.
2. Read-After-Load: 2 stalls without forwarding, 1 stall with forwarding.
3. Store word after load word: 0 stall with forwarding.
4. Branch:
5. No action taken: +3 stall for next inst (regardless of forwarding).
6. With early branching:
   1. Update before branch:
      1. +1 stall in branch inst, + 1 stall in next inst (with forwarding)
      2. +2 stall in branch inst, + 1 stall in next inst (w/o forwarding)
   2. Load before branch:
      1. +2 stalls in branch inst, + 1 stall in next inst (with forwarding)
      2. +2 stalls in branch inst, + 1 stall in next inst (w/o forwarding)
   3. Else: +1 stall for next inst (with forwarding)
7. Branch predication (with early branching)
   1. Correct pred: no stall for next inst (branch inst stall applies)
   2. Wrong pred: +1 for next inst
8. Branch predication (w/o early branching)
   1. Correct pred: no stall
   2. Wrong pred: +3 for next inst

Control dependencies

- what happpens if the instruction are branching

- the decision on when to branch is done in stage 4 (MEM), which is too late and you might have executed some intermediate instructions before branching is cnfm

- need 3 clock cycles of delay normally

solns (to decide the branch decision faster):

1. early branch (reduces to 1 delay)

- add a hardware component after reg file to check if they are the same instead of in the ALU

- HOWEVER, if the branching condition depends on an earlier inst, we cannot perform optimally,

- for data dependency, need 2 cycles of delay to wait for the prev inst result, 1 for the branch inst, 1 for the branch target

- for LW before branch, need 3 cycles, 2 for branch inst, 1 for branch target

2. branch prediction

- CAN be combined with early branching

- just assume that branch is not taken

- if guess correctly, good!

- if wrong, then just FLUSH pipeline

- 1 cycle delay for early branching

- 3 cycle delay if no early branching

3. delayed branching

- find an instruction to move to after the branch inst -> fills in the 1 cycle (using early branching) of delay incurred so tt effectively no delay

**Lect 22: Cache**

- many diff types of mem

- reg -> SRAM -> DRAM -> HDD (in order of speed)

- tradeoff between cost and speed

- principle of locality

- program will refer to small amt of memory within small time interval

- basically, make sure wtv u fetch in cache is used soon

2 types of locality (not mutually exclusive)

Temporal locality

- if item is ref, it will be ref again soon

- used in loops

Spatial locality

- if an item is ref, nearby items will be ref soon

- used in arrays

- Working set

- set of locations accessed during a time frame

- diff phase of execution might use diff working sets

AIM: capture the working set and keep it in mem close to CPU ( cache)

- Terminology

Hit: data is in cache

- hit time: time to access cache

Miss: data not in cache

-miss penalty: time to replace cache block + hit time

Hit time < Miss penalty

* Average access time = Hit rate x Hit Time + (1-Hit rate) x Miss penalty

Diagram

Description automatically generated

- Cache (SRAM)

- aims to make main memory faster

- Cache block: unit of transfer betw mem and cache

- usually 1 or more words (4 bytes in MIPS)

- every word has an address

In a 32 bit mem address

- every word in the same block has the same significant bit (dependent on block size)

- bits 31 to N is block number

- offset is the individual bytes

- bits N-1 to 0 is offset

Direct mapped cache

- take the block number % noOfCacheBlocks, will give the cache index

- take block number / noOfCacheBlocks, will give tag number (UNIQUE)

Cache block size = 2^N bytes

noOfCacheBlocks = 2^M

offset = N bits

cache Index = M bits

Tag = 32 - (N+ M) bits

How to calculate N: take block size log2

- eg if 1 block = 16 bytes, 16 bytes = 2^4 so N = 4

How calculate M: take cache size / size of block

- eg if cache = 16KB and block is 16B, then there are 2^10 blocks, M = 10

useful conversions

- 1kB = 2^10

- 1mB = 2^20

- 1GB = 2^30

Reading data

1. take the index value, go to the index in the cache

- Check the tag

- if miss, take whole block into cache (override if needed), update tag and valid bit (if needed)

- if hit, goto step 2

2. use the offset, to find the word to be sent to the processor

Writing data

- steps basically similar to reading, except when updating cache, will also need to write to

. main mem.

2 write policies:

1. write-through: write to both cache and to main mem

- goes against the intention for using cache (SLOW)

- need write buffer

2. write-back: write to main mem only when cache is replaced

- need to add an additional bit (Dirty bit)

- write operation will change dirty bit to 1

- write miss

1. write allocate: load the data into cache then write to cache (writing depends on write

policy)

2. write around: directly write to mem

**Lecture 23: Cache II**

- Block size trade off

+ takes adv of spatial locality (getting data alr stored in the cache)

- larger miss penalty (more time needed to fetch the correct data)

- block size is too big relative to cache size -> too few cache blocks -> miss rate goes up

N-way Set Associative Cache

- just like the prev direct mapped cache, except \* N for each index

- all the lines of cache is in 1 set

- Now need to have Set Index and Cache Tag

- if cache is 4kb and 4-way associative, each block is 4bytes, there are 2^10 cache blocks

-> offset N = 2 as each block = 2^2

-> since 4 way associative, number of sets = 2^10 / 4 = 2^8

-> Set Index M = 8 bits

- Now to check for hits, need to simultaneously "search" for all tags of a set

Fully Associative Cache

- no mapping function (no conflict miss)

+ can be placed in any location

- need search for all cache blocks (no more index)

- need a comparator to compare the tags

- only viable for small cache

Block replacement Policy

- for associative cache, need decide which block to replace

Least Recently Used policy

- need another data struct to keep track of the most and least recently used

- hard to keep track

Other replacement policies

- FILO

- random

- least freq used

Instruction cache:

- almost very sequential

- has very good spatial locality

- has very good temporal locality (if in a loop)