CS2100 Midterm Cheatsheet — YashyPola AY22/23 S2

Number Systems

Information Representation

Information is represented in bits, and several bits combined together form bytes (and multiples of bytes are called words, where 32-bit systems have 4-byte words and 64-bit systems have 8-byte words)
The more bits used, the more distinct values represented. In general, N bits can represent up to 2^N distinct values. Conversely, to represent a range of M values, $log_2 M$ bits is required.

Bases

In general, a base-b number $(a_n a_{n-1} a ... a_0 . f_1 f_2 ... f_m)_b$ has the value $(a_n * b^n + a_{n-1} * b^{n-1} + ... + a_0 * b^0 + f_1 * b^{-1} + f_2 * b^{-2} + ... + f_m * b^{-m})$
The point that separates the integer part and fractional part is the radix point.
For bases 10 and less, the digits from 0-9 are sufficient. Bases above 10 require additional symbols for representation. The hexadecimal system uses A,B,C,D,E,F for 10,11,12,13,14,15 respectively.

Conversion

Decimal to N-ary
Integer Conversion: Repeated Division by N (collect remainders, constructing LSB to MSB)
Fractional Conversion: Repeated Multiplication by N (collect carry, constructing MSB to LSB)
Note: Conduct Division until quotient is 0, or until out of bits. Conduct Multiplication by extracting carry, then multiplying fractional part.
Binary to Octal/Hexadecimal

1. Binary $\implies$ Octal: Partition (from radix point outwards) in groups of 3; each group corresponding to an equivalent octal digit
2. Octal $\implies$ Binary: Expand each octal digit into an equivalent group of 3 digits
3. Binary $\implies$ Hexadecimal: Partition (from radix point outwards) in groups of 4; each group corresponding to an equivalent hex digit
4. Hexadecimal $\implies$ Binary: Expand each hexadecimal digit into an equivalent group of 4 bits

Note: Can be generalized to conversion between any bases. Observation is that Octal-Binary involves $8 = 2^3$ and Hexadecimal-Binary involves $16 = 2^4$. However, general conversion involves using decimal for transitioning
N-ary to Decimal
Simply expand number as per formula in Bases section

Truncation and Rounding

The range of available digits in the number system is split into "lighter" digits and "heavier" digits. When a number needs to be shortened in terms of bits, if the nth digit is in the heavier set, the number is rounded (promoted) and if it is in the lighter set, it is truncated (chopped).

Negative Numbers

4 representations for signed binary numbers

1. Sign-and-magnitude: n-bit sign-and-magnitude scheme involves using the nth (MSB) bit as a sign-bit and (n-1) bits for magnitude. Can cover $[-(2^{n-1}-1), (2^{n-1}-1)]$ values. Invert sign bit for negation.
2. Excess: A simple translation (offset) of the binary system. For n bits, bias is usually 2^{n-1} or $2^{n-1}-1$
3. 1s complement: No difference for positive values, invert all bits to represent negative values. First bit acts as a sign bit, but remaining bits are not magnitude. Can cover $[-(2^{n-1}-1), (2^{n-1}-1)]$ values
4. 2s complement: Take 1st complement of number and then add 1. Can cover $[-2^{n-1}, (2^{n-1}-1)]$

Note: The sign-and-magnitude and 1s complement has 2 representations for 0. (00..) and (10..) for sign, and (0..) and (11..) for 1s. 2s complement has only 1 representation
Note: An n-digit base-R number X's negation can be obtained in (R-1)'s complement by $-X = R^N - X - 1$, and R's complement by dropping the -1

Arithmetic Operations

Unsigned integers

Addition, and subtraction work in the same way as decimal system.
When adding, if the values add up to a value that exceeds the range of available digits for the system, subtract the value from the base of the system and add the carryout to the next pair of values.
When subtracting, if the resulting value lies outside the available range, subtract 1 from the next pair of values and add the base of system to the result
Multiplication is just multiplying each bit to every other bit and summing the partial sums

1s complement

Conduct binary addition on numbers, and then add the carry-out to the result. Overflow occurs when numbers have same sign but result has opposite sign.

2s complement

Conduct binary addition on numbers, and discard the carry-out. Overflow occurs when numbers have same sign but result has opposite sign, or when carry-in to MSB doesn't match carry-out.
Note on Complements Operations:

1. Another way to check for overflow is if result exceeds range of representable values in the complement system.
2. Subtraction in complement systems occurs by negating values to do addition instead

IEEE-754 Floating Point

Single-precision (32-bit)

1 sign bit, 8 exponent bits, and 23 mantissa bits

Double-precision (64-bit)

1 sign bit, 11 exponent bits, and 52 mantissa bits

Fixed-point Representation

Involves assuming a radix point to split the integer and fractional parts of a number. The width of the integer part determines range of values, and the width of the fractional part determines precision. The finite nature of the fractional part hurts accuracy.

Examples

Represent -19.04492188 in Single-P IEEE

1. Convert 19.04492188 to binary
2. Shift number right until only single 1 on the left of binary point
3. Sign bit = 1, exponent bits = number of shifts (in binary), mantissa = all bits after binary point

Represent 0x3EED0000 (IEEE-754) in decimal

1. Convert to binary
2. Portion binary number into fields as per IEEE-754
3. Take mantissa, and shift by exponent amount of times to the left
4. Convert integer and fractional parts
5. Add sign as per sign bit

Represent -193.1 in fixed-point 16-bit 2s complement with 10 bits for the integer, and 6 bits for the fraction. Leave result in hexadecimal

1. Convert 193.1 to binary. Ensure fractional part is limited to 6 bits
2. Take 2s complement (ignore binary point first, then add back)
3. Convert to hexadecimal (ignore binary point when converting)

Convert $(123.321)_4$ to decimal
$1*4^2 + 2*4^1 + 3*4^0 + 4*3^{-1} + 2*4^{-2} + 1*4^{-3} = 27.890625$
Convert $(123.321)_4$ to hexadecimal
$(27.890625)_10 = 1B.E4$, obtained by converting integer and fractional parts separately and recombining. Refer Conversion section for Decimal to N-ary.

C Programming

```
// preprocessor directives
# include <stdio.h>  // header files
// constant to label. NOT a variable, NO ;
# define KMS_PER_MILE 1.609

// int, void, float, return —> reserved words
// float & double : IEEE−754 rep
int main(void) {
        float miles; // variable
                  kms;   // variable
        // functions
        printf("Enter␣distance␣in␣miles:␣")
        scanf("%f", &miles);
        // * and = are special symbols
        kms = KMS_PER_MILE * miles;

        printf("That␣equals␣%f␣km.␣\n", kms);

        return 0;
// ; terminate statement, {} denotes block
}
```

Note: A basic C program has 4 parts:

1. Preprocessor directives: header files, constants
2. Input: through stdin (using scanf) or file input
3. Compute: through operators and assignment
4. Output: through stdout (using printf) or file output

Operators

Operator Type	**Operator**
Primary Expression	$(), expr++, expr--$
Unary	$ptr, addr, -, prefix++--, (typecast)$
Binary	$*, /, mod, +-$
Assignment	$=, opr =$

Note: Increasing precedence from the bottom to the top. Primary and Binary is L-R associative, Unary and Assignment is R-L

Data Types

1. Char: 1 byte, Signed (-128 to 127), Unsigned (0 to 255)
2. Short: 2 bytes
3. Int: 4 bytes, $(-2^{31}, -2^{31} - 1)$
4. Long: 8 bytes (4-bytes for 32-bit OS)
5. Float: 4 bytes
6. Double: 8 bytes
7. Long double: 10 bytes

True/ False

1. FALSE values: false or 0 or null
2. TRUE values: everything else, true will be printed as 1

Flags

Flag	**Var Type**	**Function**
c	*char*	*printf/scanf*
d	*int*	*printf/scanf*
f	*floatordouble*	*printf*
f	*float*	*scanf*
lf	*double*	*scanf*
e	*floatordouble*	*printf(scientificnotation)*

Note: 5d : integer with a width of 5, right justified. 8.3f : real number with a width of 8, 3dp, right justified. p : prints address (for pointers). for scanf, use format specifier without indicating width, dp, etc

Pointers

1. Ampersand - address operator
2. * - declares a ptr, and dereferences (access var through ptr)
3. Incrementing variable through pointer - (*p)++, deref first then increment

```
int a, *b
*b = &a

int a, *b
b = &a

// ILLEGAL
int a, b
b = &a
```

Arrays

1. A homogenuous collection of data - store single data type
2. Stored contiguously in memory
3. An array name is a fixed (constant) pointer to the first element in the array
4. Reassignment of array is illegal

Strings

1. Array of characters
2. String functions: string.h required
3. String cannot be reassigned as well, need strcpy

```
char my_str[] = "hello";
char my_str[] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

Functions

1. strlen(s) : returns no. of chars up till terminating char
2. strcmp(s1, s2) : compares ASCII of corresponding chars, returns s1 - s2
3. strncmp(s1, s2, n) : strcmp for first n chars
4. strcpy(s1, s2) : copy s2 into s1, useful because cannot directly assign strings
5. strncpy(s1, s2, n): strcpy for first n

I/O

```
\\ in
\\ reads (size − 1) chars or until newline
fgets(str, size, stdin)
\\ reads until whitespace
scanf("%s", str)

\\out
 \\ terminates with newline (adds newline automatical
puts(str)
\\ prints until it encounters \0 in str
printf("%s\n", str)
```

Note: Don't use gets because reads until user presses enter

Structs

1. Allows grouping of heterogenuous data
2. Pass address of struct into function
3. Can be reassigned wholly unlike strings and arrays

```
typedef struct {
int length, width;
int height;
} box_t;

typedef struct {
int id;
box_t smaller_box;
} nested_box_t;

box_t mybox = {2,3,5.1};
nested_box_t big_box = {1, {4,3,6.7}};

void change_length(box_t *box) {
    \\ take note of () use to deref first
    (*box).length = 2;
}
```

Note: Syntactic sugar: (*box).length same as box $\implies$ length

Quiz Notes

1. Pass by value vs Pass by reference.
Pass by value: operation is done on a copy of the value.
Pass by reference: operation is done on value itself
2. Take note of address versus value. A pointer needs to be dereferenced to act on the value instead of address

MIPS

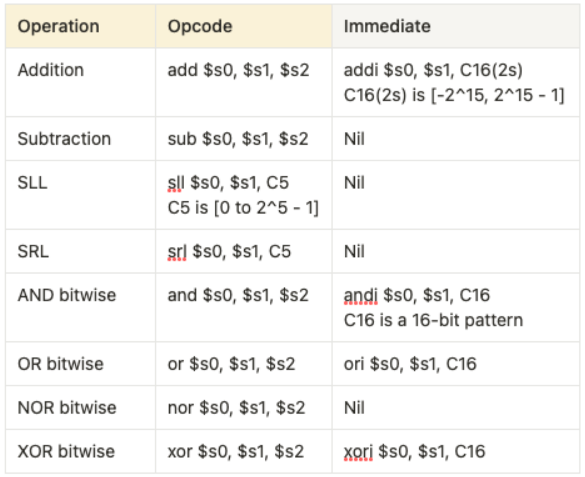
Characteristics

1. Load-store register architecture
2. 32 general-purpose small-storage registers, each 32-bit long
3. Each word contains 4 bytes
4. Memory addresses and instructions are 32-bit
5. 2^{30} memory words accessed only by memory instructions
6. Word-addressing

Major types of assembly instructions

1. Memory: move values between memory and registers
2. Calculation: arithmetic and other operations
3. Control: change the sequence of execution

Calculation



Operation	Opcode	Immediate
Addition	add \$s0, \$s1, \$s2	addi \$s0, \$s1, C16(2s) C16(2s) is [-2^15, 2^15 - 1]
Subtraction	sub \$s0, \$s1, \$s2	Nil
SLL	sll \$s0, \$s1, C5 C5 is [0 to 2^5 - 1]	Nil
SRL	srl \$s0, \$s1, C5	Nil
AND bitwise	and \$s0, \$s1, \$s2	andi \$s0, \$s1, C16 C16 is a 16-bit pattern
OR bitwise	or \$s0, \$s1, \$s2	ori \$s0, \$s1, C16
NOR bitwise	nor \$s0, \$s1, \$s2	Nil
XOR bitwise	xor \$s0, \$s1, \$s2	xori \$s0, \$s1, C16

Note: NOT operation: nor with zero register

Memory

1. lw target, offset(src) : load contents of Mem[srcAdd + offset] to target
2. sw src, offset(target) : store contents of src in Mem[targetAdd + offset]

Note: MIPS (load-store) memory is 4-byte word-addressable, i.e. addresses are in multiples of 4. This is why lw/sw instructions require offsets in multiples of 4 to ensure alignment. lb/sb instructions can have unitary offsets
Note: Byte-addressable means one byte to each location/address

Control

1. bne: take branch if NOT equal
2. beq: take branch if equal
3. j: jump unconditionally
4. slt: set to 1 if <, 0 otherwise

Bitwise Operators - Uses

1. Bit setting: or
2. Copy / mask: and
3. Alignment: shift
4. Invert: xor

Loading large constants

1. Use lui to set upper 16 bits, lower 16 autofilled with 0s
2. Use ori to set lower-order bits

Instruction Encoding

R-Format

op dest, src, tgt / shamt
add, sub, and, or, nor, slt, srl, sll

1. 6 bit opcode (always 0)
2. 3 x 5 bit registers (src, tgt, dest)
3. 1 x 5 bit shift amount
4. 6 bit function code

I-Format

op tgt, src, immediate
addi, andi, ori, slti, lw, sq, beq, bne, etc

1. 6 bit opcode
2. 2 x 5 bit registers (src, tgt)
3. 16 bit immediate

J-Format

op **partial** target address

1. 6 bit opcode
2. 26 bit immediate

Note on J-Format: 32-bit target address is possible even with only 26-bit immediate by taking 4 MSBs from PC + 4 (next instruction after jump), and omitting 2 LSBs from target address since instruction address are word-aligned. Maximum jump range = 2^{32}
General Note: All the fields are considered unsigned, need to take 2s complement for negative immediates. The reason each register is 5-bits is because MIPS has 32 registers ($log_2(32)$).

Steps

1. Check format
2. Translate opcode, registers, immediates to binary
3. Convert to hexadecimal (split into groups of 4)

PC-Relative Addressing

1. Program Counter: special register that keeps track of address of instruction being executed in processor
2. Target Address: PC + 16-bit immediate field. Can branch $\approx 2^{15}$ words = 2^{17} bytes. Can take as PC + 4 by interpreting immediate in terms of words
3. If branch is not taken, PC + 4 else (PC + 4) + (I * 4) where I is number of instructions to jump

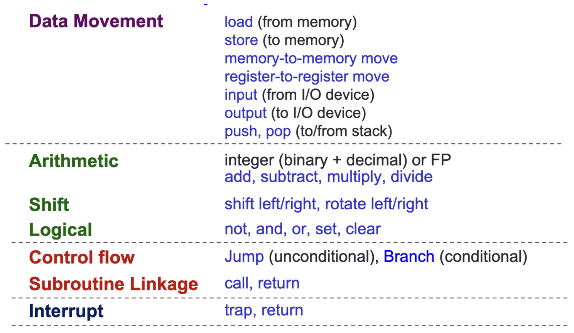
Addressing Modes

1. Register addressing: operands are registers i.e. add, sub, and
2. Immediate addressing: operand is a constant encoded in instruction itself i.e. andi, ori
3. Base addressing: operand is at the mem location whose address is the sum of a register and a constant in instruction i.e. lw, sw :
base address (specified by reg) + offset
4. PC-relative addressing: address is sum of PC and constant in the instruction i.e. beq, bne
5. Pseudo-direct addressing: 26-bit instruction concatenated with 4 MSBs of PC i.e. j

Memory Organization [General]

1. Every location has an address
2. For a k-bit address, there are 2^k locations
3. Byte-addressing: one byte difference between location addressess
4. More than one byte: word-addressing
5. Load-store architecture can only load data at word boundaries (divisible by n bytes)
6. Alignment: ensuring word boundaries are respected and maintained

Available Instructions - MIPS



Data Movement	load (from memory) store (to memory) memory-to-memory move register-to-register move input (from I/O device) output (to I/O device) push, pop (to/from stack)
Arithmetic	integer (binary + decimal) or FP add, subtract, multiply, divide
Shift	shift left/right, rotate left/right
Logical	not, and, or, set, clear
Control flow	Jump (unconditional), Branch (conditional)
Subroutine Linkage	call, return
Interrupt	trap, return

ISA

Abstract model that provides specifications of process like supported instructions, operands, etc

Storage Architectures

1. Stack: Use a stack to store operands from memory, pop when operation required. Implicit operand is ”top of stack”
2. Accumulator: Special register (implicit operand). Load values into accumulator to do operations, and store back in memory when done
3. Reg-reg: MIPS
4. Reg-mem: Load and store values directly from memory into ALU and vice versa to do operations

Endianness

The relative ordering of bytes in multiple-byte word stored in memory. Implementation specific

1. Big endian: MSB stored in lowest address (MIPS)
2. Little endian: MSB stored in highest address

Instruction Formats

Instruction Length

1. Fixed-length: easy fetch and decode, simplified pipelining and parallelism, instruction bits scarce
2. Variable: require multiple steps to fetch / decode, more flexible

Instruction Fields

1. Type and size of operands (how to divide up instruction)
2. Opcode and Operands
3. 32-bit architecture should support 8-, 16-, 32-bit int ops and 32-, 64-bit float ops

Expanding Opcode (Given Instruction Field, Calculate Max/Min Instruction Set)

Maximizing Instruction Set

1. Take maximum permutation of type with most opcode bits
2. Subtract away permutations which feature reserved opcodes from other types

Example
Type A: 8-bit opcode, 2 x 4-bit registers
Type B: 12-bit opcode, 1 x 4-bit register
Type C: 4-bit opcode, 1 x 4-bit register, 8-bit immediate

1. Allocate 0000 to Type-C, this is the only instruction handled by Type C
2. Allocate 00010000 to Type-A, this is the only instruction handled by Type A
3. Type B can now handle $2^{12} - 2^8 - 2^4 + 1 + 1$ instructions where 2^{12} is the max unrestricted permutations, 2^8 permutations have C’s reserved opcode, 2^4 permutations have A’s reserved opcode and the $+2$ adds A and C’s one allowed opcodes.

Note: The idea behind maximizing instruction set is minimizing the role of the field types that can permute the least instructions. Start from maximizing the type with the biggest opcode, and subtract away the restricted permutations

Minimizing Instruction Set

1. Take maximum permutation of type with least opcode bits
2. Add permutations which feature reserved opcodes from other types

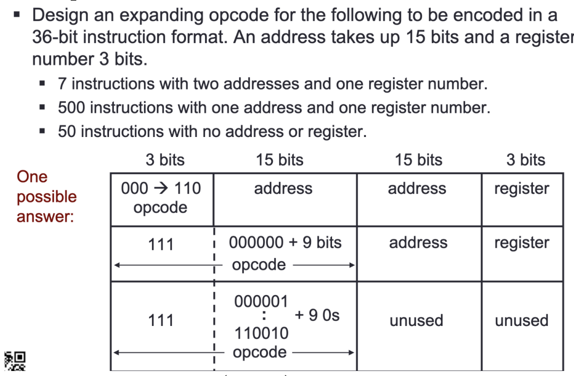
Example
Type A: 11-bit opcode, 1 x 5-bit address
Type B: 6-bit opcode, 2 x 5-bit address

1. Allocate 000000 to Type-A (Note that only need to allocate as many bits as necessary to separate from other type)
2. Type B can now handle $2^6 - 1$ instructions which makes for $(2^6 - 1) + 2^5$ total instructions

Note: The idea behind minimizing instruction set is maximize the role of the types with the smallest opcode and adding the permutations of the other type that don’t contain the restricted prefix. **Cannot just allocate 1 instruction to other type if stated that encoding space is fully used.**

Expanding Opcode (Given No. of Instructions, Derive Instruction Field Distribution)

Example



- Design an expanding opcode for the following to be encoded in a 36-bit instruction format. An address takes up 15 bits and a register number 3 bits.
  - 7 instructions with two addresses and one register number.
  - 500 instructions with one address and one register number.
  - 50 instructions with no address or register.

One possible answer:

3 bits	15 bits	15 bits	3 bits
000 → 110 opcode	address	address	register
111	000000 + 9 bits opcode	address	register
111	000001 : 110010 + 9 0s opcode	unused	unused

1. Fix 3 bit opcode (prefix) for 7 instruction type
2. Fix 9-bit prefix for 500 instruction type, and reserve it. Remaining 9 bits is for the 500 instructions
3. Allocate 6 bits in 50 instruction type to opcode, last 9 bits is set to 0

General Note

The idea behind expanding opcode is to allocate as many instructions as possible to the biggest opcode type while also ensuring types can be differentiated by fixing a prefix for the smaller types.

Datapath

General Execution Cycle

1. Instruction Fetch: Get Instruction from Memory
2. Instruction Decode: Find out operation required
3. Operand Fetch: Get operand(s) needed for operation
4. Execute: Perform required operation
5. Result Write (Store): Store result of operation
6. Repeat

Instruction Execution

Single Cycle

1. Read contents of one or more storage elements
2. Perform computation through some combinational logic
3. Write results to one or more storage elements (reg/mem)
4. All performed within a clock period

Note: Processing latency depends on longest cycle (ie slowest instruction). Parallel shorter cycles are disregarded in computing latency. Disadvantage: clock cycle must be long enough to accommodate slowest instruction

Multicycle [Not covered]

Break up instruction into several different execution steps with its own clock cycle such that time taken depends on slowest step in the particular instruction instead of all instructions taking as much as time as the slowest one (single-cycle)

MIPS-Specific

1. Fetch
2. Decode (Operation and Operands)
3. ALU (Execution)
4. Memory Stage (data read from mem)
5. Register Write
6. Repeat

Overview

Programmer (C) to Compiler (Assembly) to Assembler (Binary) to Processor. Processor then executes **datapath** (takes data from operands, processes it, writes back) and **control** (tells datapath, memory and I/O devices what to do according to opcode/func).

Fetch

1. Uses PC to fetch instruction from mem
2. Increment PC by 4 to get the next instruction (using Adder)
3. Output (to decode): instruction to be executed

Decode

1. Read opcode and determine instruction type and field lengths
2. Read data from all necessary regs
3. Output (to ALU): operation and necessary operands

ALU

1. Output (to memory): calculation result

Memory

1. Only LW/SW instructions relevant at this stage
2. Inputs (from ALU): memory address calculated and Register Value to be written to memory (only SW)
3. Output (to Register Write): result to be stored (only LW)

Register Write

1. Write the result of computation into register (irrelevant for sw, branch, jump)
2. Inputs: Destination reg number and computation result either from ALU or Memory

Elements of Datapath

Instruction Memory [1]

1. Stores instructions (sequential circuit)
2. Supplies instruction given an address;
input: instruction address M, output: instruction at address M

Adder [1.1]

1. Combinational logic to implement addition of 2 numbers

Clock [1.2]

1. Allows us to read and update the PC at the same time
2. PC is read during first half of clock period
3. PC is updated only at rising edge

Note: Overall flow - PC sends address of current instruction to Instruction Memory (which outputs that instruction). At the same time, the current instruction address is forked out to the adder where it is increased by 4 and sent back to the PC, which updates the Instruction Memory. Rinse and repeat.

Register File [2]

1. Collection of 32 registers (each 32 bits wide)
2. Input (start): Read at most 2 registers per instruction (RR1/2)
3. Write at most 1 register per instruction (WR)
4. Output: Contents of read registers (RD1/2)
5. Input (end): Data to be written (WD)

ALU [3]

1. Combinational logic to implement arithmetic and logical operations
2. Inputs: 2 32-bit numbers (operands)
3. Outputs: Result of arithmetic / logical operation
1-bit isZero signal for branching
4. Two calculations needed for branch instructions.
Branch outcome + branch target address

Data Memory [4]

1. Storage element for data of a program
2. Inputs: memory address calculated by ALU (and for SW, data to be written)
3. Outputs: Data read from memory (for LW)

Multiplexer

1. Select 1 input from multiple input lines
2. Inputs: n-lines of same width
3. Outputs: ith input line if control = i
4. Control: m bits where n = 2^m

Control path

Signals - Uses

1. RegDst: What goes in Write Register
2. RegWrite: Whether WD is written to WR
3. ALUSrc: What is operand 2 for ALU
4. MemRead: Whether anything is read from memory
5. MemWrite: Whether anything is written to memory
6. MemToReg: What result is written back to RegFile
7. PCSrc: What is PC's next instruction

Signals - Cases

Generated through opcode/func alone

1. RegDst: 0 (WR = Inst[20:16]), 1 (WR = Inst[15:11])
2. RegWrite: 0 (No Register Write), 1 (WD written to WR)
3. ALUSrc: 0 (Operand2 = Reg RD2),
   1 (Operand2 = SignExt(Inst[15:0]))
4. MemRead: 1 (Reads Memory[Target Address])
5. MemWrite: 1 (Writes Register RD2 into Mem[Target Address])
6. MemToReg: 0/1 - Write Data from ALU Result/Memory Read
7. PCSrc: 0 (PC + 4), 1 (SignExt(Inst[15:0]) << 2 + (PC + 4)).
   Set to 1 if Branch AND is0 are both 1

Control Unit

Combinational Circuit composed of AND gates, OR gates and inverters that takes in as input the 6-bit opcode and outputs the 7 control signals above + the ALUOp

1. 6-bit opcode goes through inverters placed at the necessary positions to change 0s in opcode to 1s which then go through AND gates to isolate the right instruction
2. The control signals are generated depending on where they take inputs from

ALU

32 slices of 1-bit units that each take in 4-bit ALUControl to generate a result from the operand inputs to ALU. Consists of NOT gates, AND gates, OR gates, a 2:1 multiplexer, a 4:1 multiplexer, and an adder to generate a result. There's also a Carry In (Cin) and Carry out (Cout) for the adder, that is passed between the 32 slices.

Multilevel Decoding

Alernative for the brute-force approach to generating ALUControl (use 12 bits from opcode and func). MLD involves generating 2-bit ALUOp first and combining with 4-bit func

ALUControl

4-bit signal whose MSB is Ainvert, 2nd MSB is Binvert, and last 2 bits is the ALUOp. Generated from a control block through boolean operations involving func bits and ALUOp. Control block takes in ALUOp0, ALUOp1, Func bits [2:0] as inputs.
For the actual ALU - Ainvert and Binvert allow 2:1 multiplexer to either use A/B or their negations. ALUOp allows 4:1 multiplexer to determine which result to output from ALU.

ALUOp

2-bit signal generated from opcode that consists of ALUOp1 (MSB) and ALUOp0 (LSB). 00 for lw/sw, 01 for beq, and 10 for R-type.