CS2100

AY24/25 Sem 2

by ngmh

1. Introduction

- Programming Language: A formal language that specifies a set of instructions for a computer to implement specific algorithms to solve problems
- High-Level: Level of abstraction closer to problem domain, provides productivity and portability
- Assembly Language: Textual and symbolic representation of instructions
- Machine Code: Binary bits of instructions and data
- von Neumann Architecture: Computer consisting of Input Device, Central Processing Unit (Control Unit, ALU), Memory Unit, Output Device

2. C Programming

- Edit, Compile, Execute
- C Program Format

```
preprocessor directives

main function header {
    declaration of variables
    executable statements
}
```

- Uninitialised variables do not necessarily contain zero
- Variables consist of address, name, data type, and value
- Data types: int (4 bytes), float (4 bytes), double (8 bytes), char (1 byte)
- C is strongly typed
- Preprocessor Directives: Inclusion of header files, Macro Expansions
- Input / Output : scanf, printf, remember address for scanf
- Format Specifier: c (char), d (int), f (float), lf (double), e (float or double)
- $\%n.mf$ means width n including m decimal places
- %% for literal %
- Boolean expressions can be short-circuited

3. Number Systems

- Data is stored as bits
- Byte: 8 bits, Nibble: 4 bits, Word: Multiple of Bytes
- N bits can represent 2^N values, M values require $\lceil log_2 M \rceil$ bits
- Weighted-Positional Number System: Left is increasing powers, Right is decreasing powers
- Number Systems: Decimal (Base 10), Binary (Base 2 / 0b), Octal (Base 8 / 0), Hexadecimal (Base 16 / 0x)
- Base-R to Decimal: Multiplication by weights and summation
- Decimal to Binary Conversion: Repeated Multiplication / Division by 2 (Works for other bases as well)
- Repeated Divison by 2:

```
43 (10) = 101011 (2)
21 r 1 LSB
10 r 1
5 r 0
2 r 1
1 r 0
0 r 1 MSB
```

- Repeated Multiplication by 2:

```
0.3125 (10) = 0.0101 (2)
0.625 c 0 MSB
1.250 c 1
0.500 c 0
1.000 c 1 LSB
```

- For conversion with other bases, use Base-10 as intermediary
- Between bases of multiples of 2, simplify partition or split characters
- ASCII: 7 bits + 1 parity bit (used as checksum)
- Odd Scheme: Parity after adding parity bit must be odd
- Unsigned Numbers: Only non-negative values (as opposed to signed)
- Overflow: Result of addition or subtraction goes out of range, can be detected by checking if sign of result matches

Sign-and-Magnitude

- 1 sign bit (MSB), other bits for magnitude
- Sign: 0 is positive, 1 is negative
- Range: $-(2^{n-1}-1)$ to $2^{n-1}-1$
- Has redundant zero
- Negation: Just invert sign bit

1s Complement

- $-x = 2^n - x - 1$
- Sign: See MSB
- Negation: Invert all bits
- Range: $-(2^{n-1}-1)$ to $2^{n-1}-1$
- Has redundant zero
- Addition: Perform binary addition, adding 1 to the result if there is carry out due to redundant zero
- Subtraction: Perform addition with negation of the second term

2s Complement

- $-x = 2^n - x$
- Sign: See MSB
- Negation: Invert all bits, then add 1
- Range: -2^{n-1} to $2^{n-1}-1$
- No redundant zero
- Addition: Perform binary addition, ignoring carry out
- Subtraction: Perform addition with negation of second term

Excess Representation

- Allows range of values to be distributed evenly between positive and negative values by a simple translation
- Value v is represented as $v+n$ in Excess n
- Range: $-n$ to $n-1$
- For even distributio of k-bit numbers we should use Excess-2^{k-1} or Excess-$2^{k-1}-1$

Radix Complement

- For number of digits n and radix b
- $(b-1)$s Complement: $-x = b^n - x - 1$
- (b)s Complement: $-x = b^n - x$

Fraction Extension

- For n bits and f fractional bits
- 1s Complement: $-x = 2^n - x - 2^{-f}$, invert all bits
- 2s Complement: $-x = 2^n - x$, invert all bits, add 2^{-f} (the smallest bit)
- Resolution is the smallest bit value
- Might have to round off if non-exact

Real Numbers

- Fixed Point Representation: Number of bits allocated for whole number part and fractional part
- IEEE754 Floating Point Representation:
  - Sign, Exponent, Mantissa (Fraction)
  - Single Precision: 1-bit sign, 8-bit exponent in excess-127, 23-bit mantissa
  - Double Precision: 1-bit sign, 11-bit exponent in excess-1023, 52-bit mantissa
  - Sign: 0 for positive, 1 for negative
  - Mantissa is normalised, with implicit leading bit 1 (e.g. $110.1 = 1.101 \times 2^2$)
  - Exponent is stored in excess-127 (e.g. $2 + 127 = 129$
  - Merge components together and express in hexadecimal

4. Pointers and Functions

Pointers

- The address of a variable can be found with &, and has format specifier p
- Pointers store these addresses, along with the data type they are pointing to
- They are defined using *
- Dereference pointers using *
- Incrementing a pointer moves its value by the data type size

Functions

- User-Defined Functions: Require prototype before main, and definition after main
- Function Prototype: Return type, method name, types of parameters (name optional)
- Use void in prototype of function with no parameters, empty means unspecified
- C parameters are pass by value
- Scope Rule: Local variables are only accessible within function they are declared
- Can use static to make values persistent across calls
- Function can take in pointers instead to achieve pass by reference

5. Arrays, Strings and Structures

Arrays

- Homogeneous collection of data
- Declared by element type, array name, and size
- Elements are 0-indexed
- Arrays can be initialised only at declaration
- Array name is a fixed constant pointer and cannot be altered
- Address and value of array variable are always the same
- In function parameters, need to specify [] to show it is an array or simplify take in a pointer instead
- Functions involving arrays need size parameters as arrays decay into pointers when passed as parameters by array name

Strings

- A string is an array of characters terminated by the null character \0
- Can be initialised directly with a string
- Input: fgets(str, size, stdin) or scanf("%s", str)
- Output: puts(str) or printf("%s", str)
- fgets() also reads in the newline character, so we might have to replace it
- puts() automatically adds a newline
- Some string functions include strlen(), strcmp(), strncmp(), strcpy(), strncpy()
- A non-null terminated string might cause illegal access of memory

Structures

- Allow grouping of heterogeneous members

```
typedef struct {
    int acctNum;
    float balance;
} account_t;
```

- Type declaration, not a variable
- Initialisation is similar to arrays
- Members are accessed using .
- Assignments are okay, unlike arrays
- Structs can also be passed into functions by value and returned from functions
- Entire contents even arrays are copied when they are passed by value
- Can use arrow operator $->$ instead of $(*)$.

7. MIPS Introduction

Overview

- Instruction Set Architecture: Abstraction on the interface between hardware and the low-level software
- Machine Code: Instructions in binary / hexadecimal, Hard and tedious to code
- Assembly: Human readable, Easier to write than machine code, symbolic version, may also provide pseudo-instructions as syntactic sugar
- A computer has processor and memory, with a bus acting as the bridge between them
- Code and data reside in memory, to be transferred into the processor during execution
- To avoid having to frequently access memory, the processor has temporary storage for values known as registers
- Need memory instructions to move data between registers as well as to and from memory
- Need arithmetic instructions as well, some of which involve constants
- Need instructions to manage control flow

Registers

- Processor has fast memory in the form of registers
- Typical architecture has 16 to 32 registers, which the compiler associates with variables
- Registers have no data type
- MIPS has 32 registers (refer to Green Card)

Language and Basic Operations

- Each instruction executes a simple command
- Each line has at most 1 instruction
- Use # for comments
- Operation followed by destination then sources
- e.g. add $rd, $rs, $rt
- Immediate operation: Has an immediate instead of second argument which uses 16 bit 2s complement
- Immediate is sign extended, meaning the MSB is duplicated all the way across the upper half, which is actually value preserving
- subi does not exist, use addi with negative constant
- Register $zero is guaranteed to have a zero value
- Bit shifting is limited to 5 bits as registers are only 32 bits
- not does not exist, use nor $t0, $t0, $zero instead
- To load large constants, we need to use lui to set the upper bits, and ori to set the lower bits
- Basic Operations:

Operation	Opcode in MIPS	Meaning
Addition	`add $rd, $rs, $rt`	`$rd = $rs + $rt`
	`addi $rt, $rs, C16`$_{2s}$	`$rt = $rs + C16`$_{2s}$
Subtraction	`sub $rd, $rs, $rt`	`$rd = $rs - $rt`
Shift left logical	`sll $rd, $rt, C5`	`$rd = $rt << C5`
Shift right logical	`srl $rd, $rt, C5`	`$rd = $rt >> C5`
AND bitwise	`and $rd, $rs, $rt`	`$rd = $rs & $rt`
	`andi $rt, $rs, C16`	`$rt = $rs & C16`
OR bitwise	`or $rd, $rs, $rt`	`$rd = $rs | $rt`
	`ori $rt, $rs, C16`	`$rt = $rs | C16`
NOR bitwise	`nor $rd, $rs, $rt`	`$rd = $rs ↓ $rt`
XOR bitwise	`xor $rd, $rs, $rt`	`$rd = $rs ^ $rt`
	`xori $rt, $rs, C16`	`$rt = $rs ^ C16`

8. More Instructions

Memory Instructions

- The main memory can be viewed as a large single dimension array
- Each location of the memory has an address
- We can use addresses to access a single byte / word
- A word usually has a size which is a power of 2, and usually coincides with register size
- Word Alignment: Words that are aligned in memory begin at a byte address which is a multiple of word size
- lw / sw $t0, 4($s0)
- Address is offset from a base address, and must be word aligned

Control Flow

- Conditional: Branching
- Unconditional: Jumping
- beq $r1, $r2, label
- Jump to statement with label if values in registers are equal
- Labels represent instruction addresses and are not instructions
- To produce shorter code, invert the condition in C code to favor early exit
- Use combination of branching and jumps to make a for loop

```
        add $s0, $zero, $zero
        addi $s1, $zero, 10
    Loop: beq $s0, $s1, Exit
        addi $s2, $s2, 5
        addi $s0, $s0, 1
        j Loop
    Exit:
```

- Use slt (set less than) to simulate comparisons

9. MIPS Instruction Formats and Encoding

- Every MIPS instruction is 32 bits long
- Register Format: op $r1, $r2, $r3
- Immediate Format: op $r1, $r2, Immd
- Jump Format: op Immd
- Refer to Green Card for more information

R-Format

- opcode: Specifies instruction, 0 for all R-format instructions
- funct: Combines with opcode to specify instruction
- rs: Specifies source register
- rt: Specifies target register
- rd: Specifies destination register
- shamt: Bit shift amount

I-Format

- Bigger field for intermediate values
- rt is destination register instead as there is no rd
- immediate: 16 bits signed integer in 2s complement

Addressing

- Since instructions are stored in memory, they also have addresses which are word aligned
- Program Counter: Special register that keeps address of instruction being executed in processor
- Conditional instructions are I-format and use immediate for PC-relative addressing
- Immediate is specified as target address relative to PC, as a number of words
- If branch is not taken, $PC = PC + 4$
- If branch is taken, $PC = (PC + 4) + (Immd \times 4)$
- Start counting from next line of code

Addressing Modes

- Register Addressing: Operand are all registers
- Immediate Addressing: Operand is a constant within instruction itself
- Base Addressing / Displacement Addressing: One of the register operands is a memory location while another immediate value corresponds to the offset from the given memory location
- PC-Relative Addressing: Base address is given by PC register
- Pseudo-Direct Addressing: Use only upper 4-bits of PC register, with rest of address being obtained from instruction, with last 2 bits 00 due to word alignment

J-Format

- Allows jumps to further locations through psuedo-direct addressing
- Target address field is only 26 bits however
- But since instructions are word aligned, we can assume the last 2 bits to be 00
- The remaining 4 bits are taken from the MSB of PC+4
- Jump boundary is now 256MB

10. Instruction Set Architecture

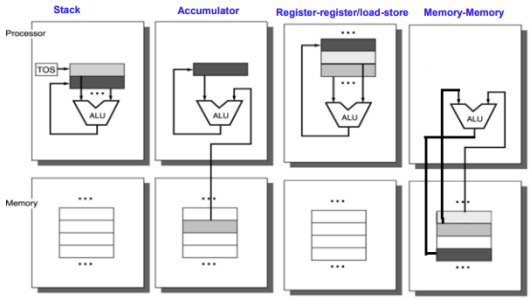
- Complex Instruction Set Computer (CISC): Single instruction performs complex operation, smaller program size, but complex implementation which does not leave room for hardware optimisation (e.g. x86-32)
- Reduced Instruction Set Computer (RISC): Simple instruction set, easier to optimise hardware, but burden is on software to implement high-level language statements (e.g. MIPS, ARM)

Data Storage

- Storage Architecture: How are operands and computation results stored
- Stack Architecture: Operands are implicitly on top of stack
- Accumulator: One operand is implicitly in the accumulator (a special register)
- General-Purpose Register Architecture: Only explicit operands (Register-Memory, Register-Register / Load-Store)
- Memory-Memory Architecture: All operands in memory
- The most common is general-purpose register, with RISC using Register-Register design and CISC using a mix of both

Stack	Accumulator	Register (load-store)	Memory-Memory
Push A	Load A	Load R1,A	Add C, A, B
Push B	Add B	Load R2,B	
Add	Store C	Add R3,R1,R2	
Pop C		Store R3,C	

C = A+B



Memory Addressing Mode

- Give k-bit addresses, the address space has size 2^k with each memory transfer consisting of one word of n bits
- Endianness: Relative ordering of bytes in a multiple-byte word store in memory
- Big-Endian: MSB stored in lowest address (e.g. MIPS)
- Little-Endian: LSB stored in lowest address
- Addressing Modes in MIPS: Register, IMmmediate, Displacement

Operations

- Standard Operation Types:
  - Data Movement
  - Arithmetic, Shift, Logical
  - Control Flow, Subroutine Linkage
  - Interrupt
  - Synchronisation, String, Graphics
- Frequently used instructions should be made the fastest

Instruction Formats

- Instruction Length
  - Variable-length: Require multi-step fetch and decode, more flexibility but more complex
  - Fixed-length: Used in most RISCs, allow for easy fetch and decode, simplifies pipelining and parallelism, but instruction bits are scarce
  - Hybrid: Mix of both
- Instruction Fields
  - Type and Size of operands
  - Consists of opcode and operands
  - Typical type and sizes include characters (1 byte), word, floating points with single and double precision (1 / 2 words)

Encoding the Instruction Set

- Instruction Encoding
  - How are instructions represented in binary format
  - Variable v/s Fixed v/s Hybrid
  - Number of registers, adressing modes, number of operands
- Encoding Fixed Length Instructions
  - Expanding Opcode Scheme: Opcode has variable lengths for different instructions
  - Example: 16 bit instructions, 2 types of instructions
  - Type A: 2 operands, each 5 bits
  - Type B: 1 operand, 5 bits
  - Maximum Number: Minimise Type A, $1 + (2^6 - 1) \times 2^5 = 2017$
  - Minimum Number: Maximise Type A, $(2^6 - 1) + 1 \times 2^5 = 95$

11. MIPS Datapath

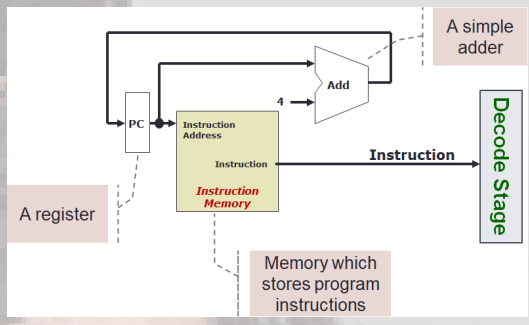
- Processor has Datapath: Collection of components that process data and performs arithmetic, logical, and memory operations
- It also has Control: Tells the datapath, memory, and I/O devices what to do according to program instructions
- Basic Instruction Execution Cycle
  - Fetch: Get instruction from memory, address is in PC
  - Decode: Find out the operation required
  - Operand Fetch: Get operands needed for operation
  - Execute: Perform the required operation
  - Result Write (Store): Store the result of the operation
- MIPS Instruction Execution
  - Decode and Operand Fetch are merged as decoding is simple
  - Execute is split into ALU (Calculation) and Memory Access

Building a MIPS Processor

Fetch Stage

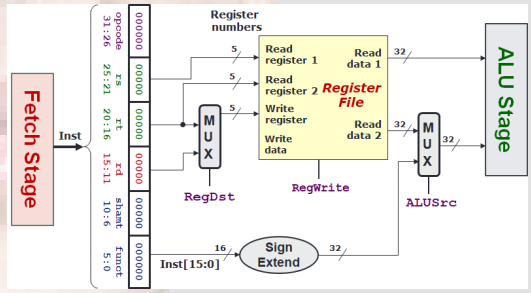
- Use PC to fetch instruction from memory
- Increment PC by 4 to get address of next instruction
- Output the instruction to be executed to the decode stage

- Instruction Memory:
  - Storage element for instructions, functions like a big array
  - Is a sequential circuit
  - Has internal state that stores information
  - Clock signal is assumed and not shown
  - Supplies instruction when given an address
- Adder: Combinational logic to implement addition of two numbers, has no state
- Clocking: Read and update PC at the same time by performing each action at different parts of the clock cycle, update is only performed at rising edge



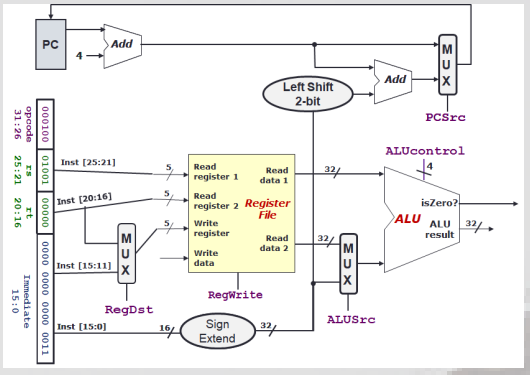
Decode Stage

- Gather data from instruction fields: opcode and data from necessary registers
- Input instruction is taken from fetch stage
- Output operation and necessary operands to ALU stage
- Register File:
  - Collection of 32 registers
  - Each register is 32 bit, and at most two are read and one written
  - RegWrite: Control signal to indicate writing of register (1 is write, 0 is no write)
- Problems
  - Not all instructions have destination in the same place
  - Multiplexer and RegDst control signal is used to determine where to read from
  - Read Data 2 might be an immediate value instead of a register input
  - Multiplexer is used to choose the correct operand, with sign extension applied to 16-bit immediate to make it 32-bit
- Multiplexer:
  - Selects one input from multiple input lines
  - Input: n lines of same width
  - Control: m bits where $n = 2^m$
  - Output: Select the i^{th} input line based on control



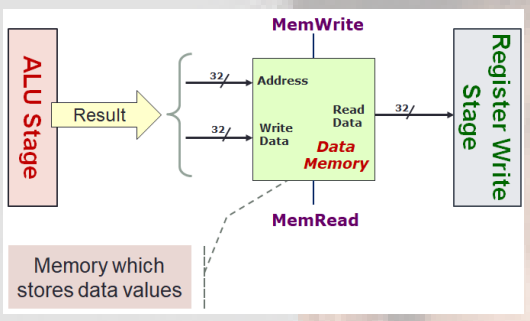
ALU Stage

- Arithmetic Logic Unit
- a.k.a Execution Stage
- Perform real work for most instructions
- Input operation and operands are from decode stage
- Output calculation result is given to memory stage
- ALU
  - Combinational logic to implement arithmetic and logical operations
  - Input: 2 32-bit numbers
  - Control: 4-bit signal
  - Output: Result of operation and 1-bit signal to indicate whether result is 0
- Branch Instructions
  - Branch Outcome: Determine using ALU for comparison and isZero? signal
  - Branch Target Address: Introduce logic to calculate the address using PC from fetch stage and Offset from decode stage



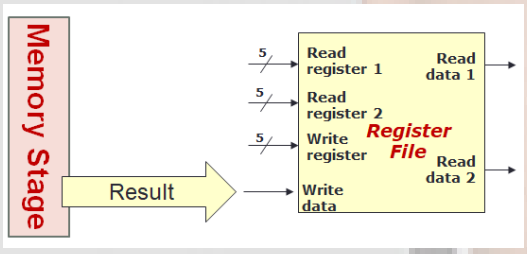
Memory Stage

- Only load and store operations are needed
- Use memory address calculated by ALU stage and read or write data memory
- All other instructions remain idle
- Input is computation result as memory address (if applicable) from ALU stage
- Output is result to be stored (if applicable) to register write stage
- Data Memory
  - Storage element for data of program, functions like a big array
  - Input: Memory Address, Data to be Written
  - Control: Read and Write controls, only one at a time
  - Output: Data read from memory for load instructions
- Need Read Data 2 from decode stage as Write Data
- MemToReg: Control signal to indicate whether result came from memory or ALU unit, inverted

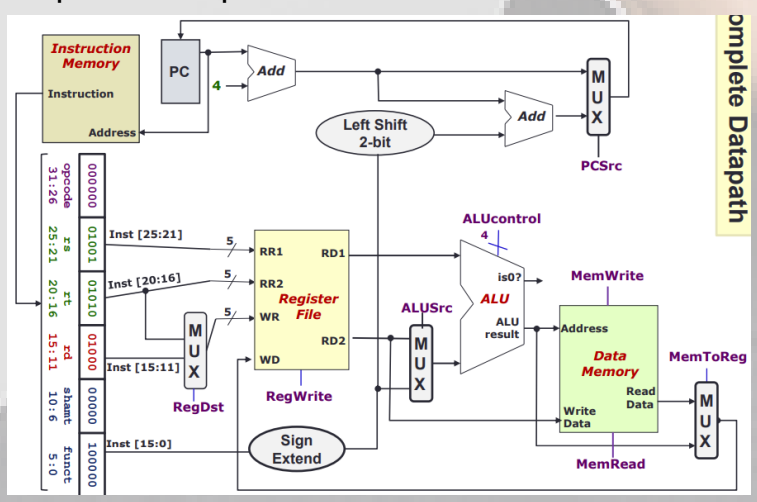


Register Write Stage

- Most instructions write the result of computation into a register
- Need destination register number and computation result
- Exceptions are stores, branches, and jumps
- Input is the computation result either from memory or ALU
- No additional elements, simply route correct result into register file, using Write Register number generated back in decode stage



Complete MIPS Datapath

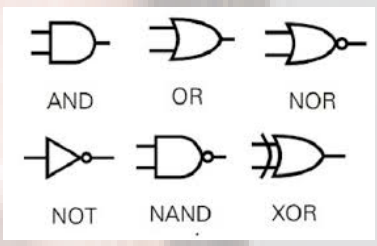


Compiling and Execution

- Compiler translate to high-level language to assembly language
- Assembler translates to machine code
- Processor executes the machine code
- Compiling to MIPS
  - Compilation is structured, and each structure can be compiled independently
  - Variable-to-Register Mapping
  - Conditions can be inverted for shorter code
  - Complex operations should be broken down with temporary registers
  - Array access is lw while array update is sw
  - Remember that word addresses differ by word size

12. MIPS Control

- Control signals are generated based on the instruction to be executed
- A control unit is needed to design a combinational circuit to generate signals based on opcode and possibly funct
- General Flow: Take note of instructions to be implemented, go through each signal to observe how it is generated, construct truth table, then design control unit using logic gates
- Logic Gates

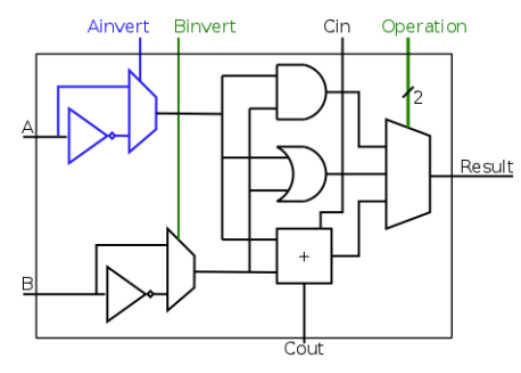


Control Signals

- RegDst:
  - Selects destination register number
  - False: False: Write Register = rt / Inst[20:16]
  - True: Write Register = rd / Inst[15:11]
- RegWrite:
  - Enable writing of register
  - False: No register write
  - True: New value will be written
- ALUSrc:
  - Select second operand for ALU
  - False: Operand2 = Register Read Data 2
  - True: Operand2 = Immd / SignExt(Inst[15:0])
- MemRead / MemWrite
  - Enable reading / writing of data memory
  - False: Not performing memory read access or memory write
  - True: Read memory using Address / write register read data 2 to memory at address
- MemToReg:
  - Select result to be written back to register file
  - True: Register Write Data = Memory Read Data
  - False: Register Write Data = ALU Result
- PCSrc:
  - Select the next PC value
  - Use isZero? signal from ALU to determine branch outcome
  - If instruction is a branch AND taken then true
  - False: PC = PC + 4
  - True: PC = Immd « 2 + (PC + 4)
- : ALUControl
  - See below

ALUcontrol

- Select the operation to be performed
- Simplified ALU
  - 1-bit ALU requires 4 control bits
  - Since MIPS is 32-bit, 32 ALUs are cascaded together
  - Cin is carry in, Cout is carry out
  - Cout of each ALU is connected to Cin of next ALU
  - This allows us to add the full 32-bit range of numbers
  - Square is a full adder
  - Trapezium represents multiplexers



- ALUcontrol signal is sent to control ALU
- Subtract can be implemented as $A + B' + 1 = A + (-B)$, with the $+1$ coming from Cin
- NOR is implement as $A' \wedge B' = (A \vee B)'$ by De Morgan's law

ALUcontrol			Function
Ainvert	Binvert	Operation	
0	0	00	AND
0	0	01	OR
0	0	10	add
0	1	10	subtract
0	1	11	slt
1	1	00	NOR

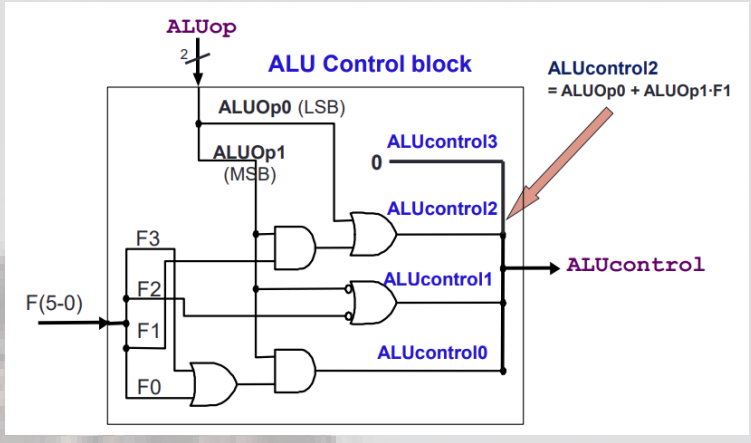
- Multilevel Decoding
  - ALUcontrol depends on 6-bit opcode and 6-bit funct
  - Brute Force: Use both directly and find expressions with 12 variables
  - Multilevel Decoding: Use some input to reduce cases, then generate the full output

ALUop

- Use opcode to generate 2-bit ALUop signal
- lw / sw : 00 (both use add)
- beq: 01 (uses sub)
- R-type: 10 (uses funct)
- Use ALUop and funct to generate 4-bit ALUcontrol signal
- We can ignore some inputs that don't give us any useful information such as F5 and F4

	ALUop		Funct Field (F[5:0] == Inst[5:0])						ALU control
	MSB	LSB	F5	F4	F3	F2	F1	F0	
lw	0	0	X	X	X	X	X	X	0 0 1 0
sw	0	0	X	X	X	X	X	X	0 0 1 0
beq	~~0~~ X	1	X	X	X	X	X	X	0 (1) 1 0
add	1	~~0~~ X	~~1~~ X	~~0~~ X	0	0	0	0	0 0 1 0
sub	1	~~0~~ X	~~1~~ X	~~0~~ X	0	0	1	0	0 (1) 1 0
and	1	~~0~~ X	~~1~~ X	~~0~~ X	0	1	0	0	0 0 0 0
or	1	~~0~~ X	~~1~~ X	~~0~~ X	0	1	0	1	0 0 0 1
slt	1	~~0~~ X	~~1~~ X	~~0~~ X	1	0	1	0	0 (1) 1 1

- beq is uniquely determined by ALUop 1 = 1
- F5 is always 1 for R-format, F4 is always 0
- Possible combinational logic:
- ALUcontrol 3 = 0
- ALUcontrol 2 = $ALUop0 \vee (ALUop1 \wedge F1)$
- ALUcontrol 1 = $(\neg ALUop1) \vee (ALUop1 \wedge \neg F2) = \neg ALUop1 \vee \neg F2$ by De Morgan's law
- ALUcontrol 0 = $(F0 \vee F3) \wedge ALUop1$
- This gives us the following combinational circuit:



Control Circuit Outputs

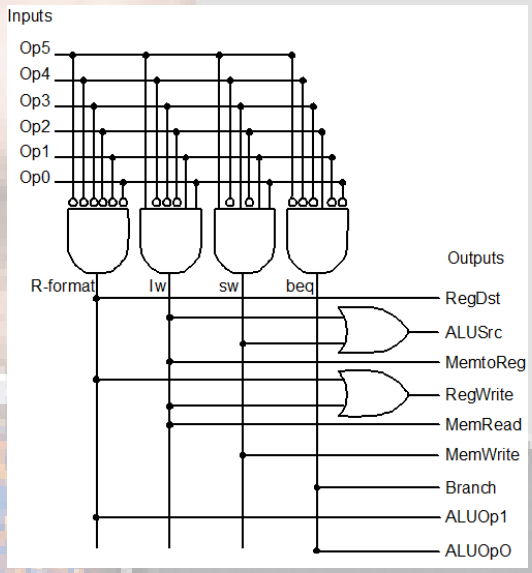
	RegDst	ALUSrc	MemTo Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop	
								op1	op0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Control Inputs

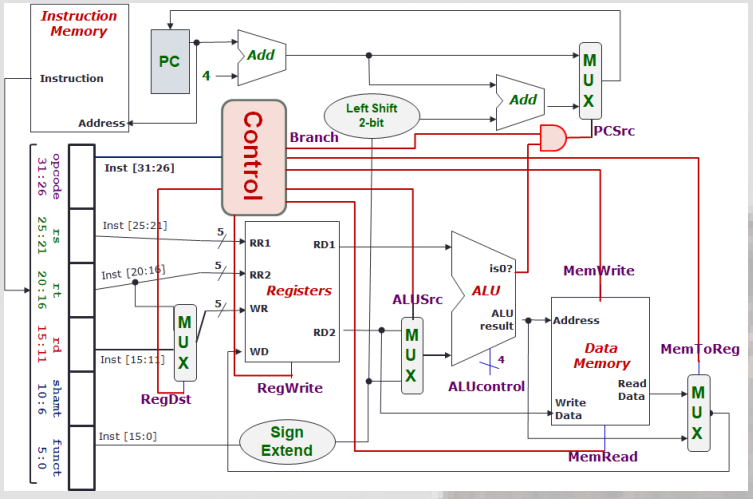
	Opcode (Op[5:0] == Inst[31:26])						
	Op5	Op4	Op3	Op2	Op1	Op0	Value in Hexadecimal
R-type	0	0	0	0	0	0	0
lw	1	0	0	0	1	1	23
sw	1	0	1	0	1	1	2B
beq	0	0	0	1	0	0	4

Control Circuit

- Small circles are inverters
- Input are combined using AND gates to determine output of signal based on above tables



Finished Datapath and Control



Instruction Execution

- Read contents of one or more storage elements
- Perform computation through combinational logic
- Write results to one or more storage elements
- All the steps are performed within a clock period
- Writing must be performed during rising edge
- Single Cycle Implementation
  - All instructions will take as much time as the slowest one
- Multicycle Implementation
  - Break instructions up into execution steps
  - Each execution step takes one clock cycle
  - Cycle time is shorter, clock frequency is higher
  - Instructions take variable number of clock cycles to complete execution
- Pipelining
  - Break up the instructions into execution steps, one per clock cycle
  - Allow different instructions to be in different execution steps simultaneously