# 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
- $1$s Complement: $-x = 2^n - x - 2^{-f}$, invert all bits
- $2$s 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

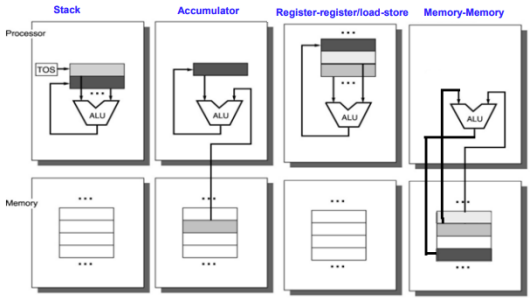
- Complex Instruction Set Computer (CISC): Single instruction performs complex operation, smaller program size, but complex implementation which does not leave room for hardware optimisation (e.g. x86-32)
- Reduced Instruction Set Computer (RISC): Simple instruction set, easier to optimise hardware, 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

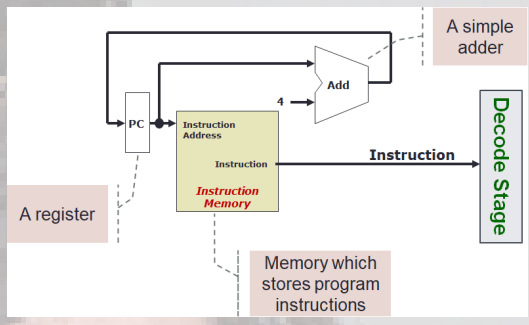
- Processor has Datapath: Collection of components that process data and performs arithmetic, logical, and memory operations
- It also has Control: Tells the datapath, memory, and I/O devices what to do according to program instructions
- Basic Instruction Execution Cycle
  - Fetch: Get instruction from 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

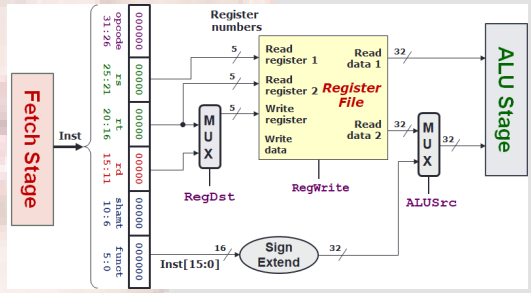
- Use PC to fetch instruction from memory
- Increment PC by 4 to get address of next instruction
- Output the instruction to be executed to the decode stage

- 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



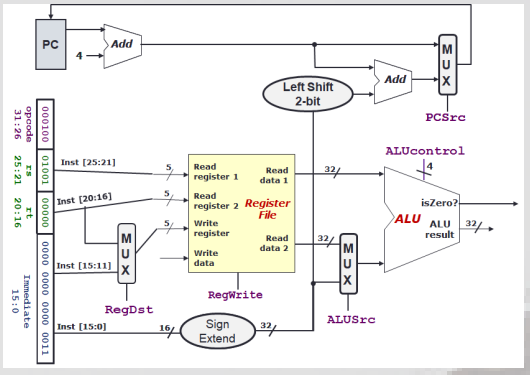

## Decode Stage

- 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



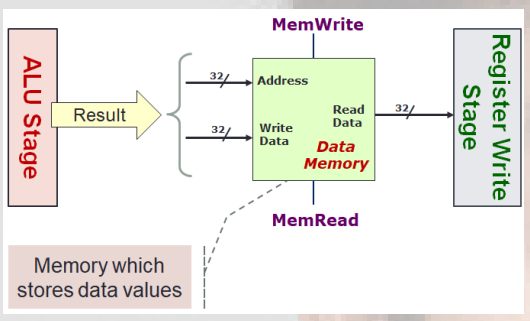

## ALU Stage

- 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



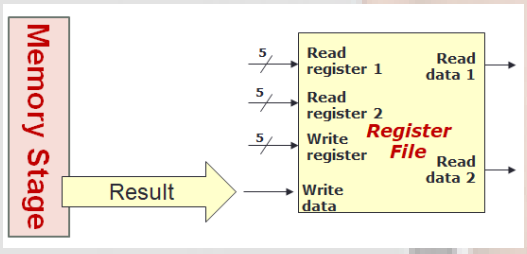

## Memory Stage

- 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

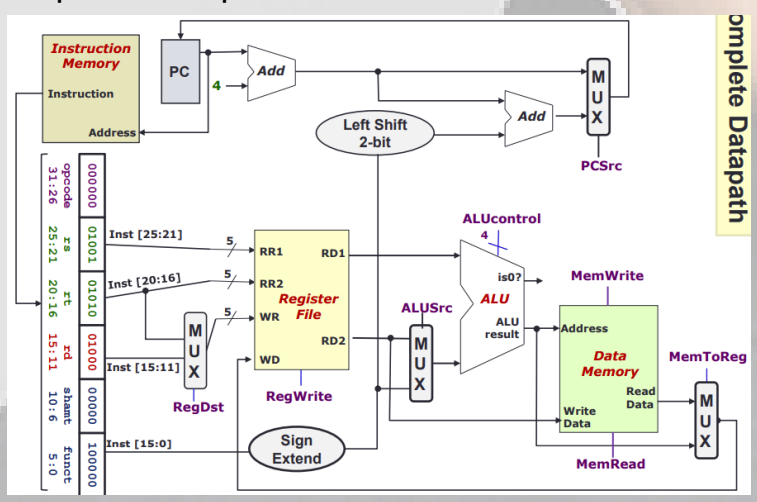



## Register Write Stage

- 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

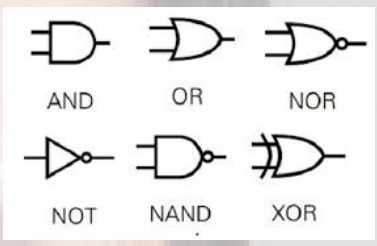



## Compiling and Execution

- 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

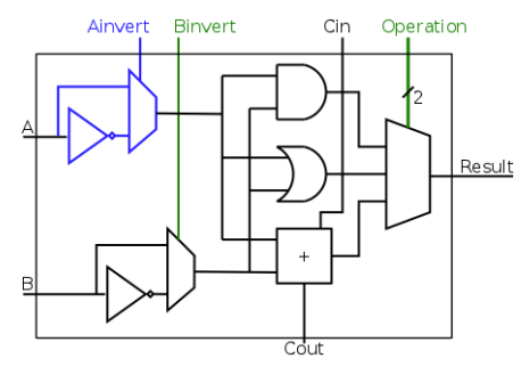



## Control Signals

- 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 |

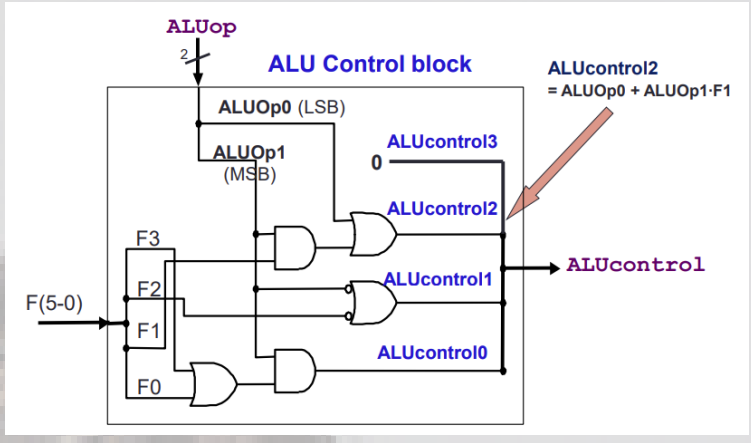
- Multilevel Decoding
  - ALUcontrol depends on 6-bit opcode and 6-bit funct
  - Brute Force: Use both directly and find expressions with 12 variables
  - Multilevel Decoding: Use some input to reduce cases, then generate the full output

## 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

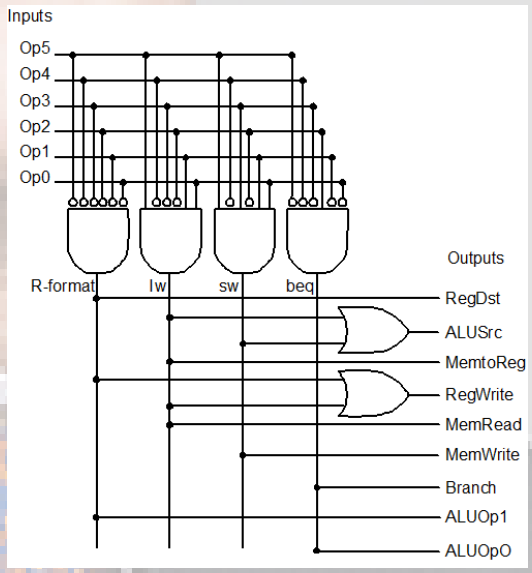
| | RegDst | ALUSrc | MemTo Reg | Reg Write | Mem Read | Mem Write | Branch | ALUop | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | op1 | op0 |
| R-type | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| lw | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| sw | X | 1 | X | 0 | 0 | 1 | 0 | 0 | 0 |
| beq |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

## Control Inputs

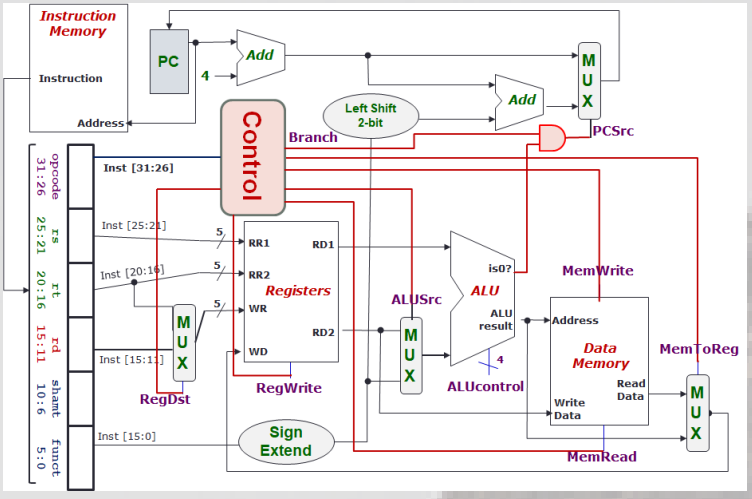
| | Opcode ( Op[5:0] == Inst[31:26] ) | | | | | | |
|---|---|---|---|---|---|---|---|
| | Op5 | Op4 | Op3 | Op2 | Op1 | Op0 | Value in Hexadecimal |
| R-type | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| lw | 1 | 0 | 0 | 0 | 1 | 1 | 23 |
| sw | 1 | 0 | 1 | 0 | 1 | 1 | 2B |
| beq | 0 | 0 | 0 | 1 | 0 | 0 | 4 |

## 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




- RR1: rs or Inst[25:21]
- RR2: rt or Inst[20:16]
- WR: Selected from MUX
- WD: Result of final MUX
- Opr1: RD1 = R[rs]
- Opr2: Selected from either Immd or RD2=R[rt]
- Address: ALU result
- Write Data: RD2 = R[rt]

### 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

## 13. Boolean Algebra

- High/True/1 and Low/False/0
- Combinational Circuit: No memory, output depends solely on input
- Sequential Circuit: With memory, output depends on both input and current state
- AND: $\cdot$, OR: $+$, NOT: $'$
- Precedence: NOT > AND > OR

### Laws / Theorems

- Identity Laws: $A+0=0+A=A$, $A\cdot 1=1\cdot A=A$
- Inverse/Complement Laws: $A+A'=A'+A=1$, $A\cdot A'=A'\cdot A=0$
- Commutative Laws: $A+B=B+A$, $A\cdot B=B\cdot A$
- Associative Laws: $A+(B+C)=(A+B)+C$, $A\cdot(B\cdot C)=(A\cdot B)\cdot C$
- Distributive Laws: $A\cdot(B+C)=(A\cdot B)+(A\cdot C)$, $A+(B\cdot C)=(A+B)\cdot(A+C)$
- Duality: If AND/OR and 0/1 in an equation are swapped, it remains valid
- Idempotency: $X+X=X$, $X\cdot X=X$
- One / Zero Element: $X+1=1+X=1$, $X\cdot 0=0\cdot X=0$
- Involution: $(X')'=X$
- Absorption 1: $X+X\cdot Y=X$, $X\cdot(X+Y)=X$
- Absorption 2: $X+X'\cdot Y=X+Y$, $X\cdot(X'+Y)=X\cdot Y$
- DeMorgan's (Can be generalised): $(X+Y)'=X'\cdot Y'$, $(X\cdot Y)'=X'+Y'$
- Consensus: $X\cdot Y+X'\cdot Z+Y\cdot Z=X\cdot Y+X'\cdot Z$, $(X+Y)\cdot(X'+Z)\cdot(Y+Z)=(X+Y)\cdot(X'+Z)$

### Standard Forms

- Literal: Boolean variable on its own or in its complemented form
- Product Term: A single literal or a logical product of several literals
- Sum Term: A single literal or a logical sum of several literals
- Sum-of-Products Expression: A product term or a logical sum of several product terms
- Product-of-Sums Expression: A sum term or a logical product of several sum terms
- Minterm: Product term that contains n literals from all the variables
- Maxterm: Sum term that contains n literals from all the variables
- Minterm is complement of maxterm
- Canonical Form: Unique form of representation
- Sum of Minterms: Gather minterms of function where output is 1
- Product of Maxterms: Gather maxterms of function where output is 0
- Opposite of each other

| x | y | Minterms | | Maxterms | |
|---|---|---|---|---|---|
| | | Term | Notation | Term | Notation |
| 0 | 0 | x'·y' | m0 | x+y | M0 |
| 0 | 1 | x'·y | m1 | x+y' | M1 |
| 1 | 0 | x·y' | m2 | x'+y | M2 |
| 1 | 1 | x·y | m3 | x'+y' | M3 |

## 14. Logic Circuits

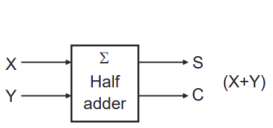
- See previous pages for list of logic gates
- XOR: Check different, XNOR: Check same
- Fan-in: Number of inputs to a gate
- Cannot have hanging inputs!
- Draw circle for wire intersection
- Complete set of logic: Can make any boolean function
- e.g. {AND, OR, NOT}, {NAND}, {NOR}
- NAND with both inputs $X$ gives $X'$
- NAND of NAND(X, Y) gives $X\cdot Y$
- NAND (X', Y') gives $X+Y$
- Similar for NOR but swap some cases around
- SOP: Use 2 level AND-OR or NAND circuit
- POS: Use 2 level OR-AND or NOR circuit

## 15. Simplification

- Minimise number of literals
- Algebraic Simplification can be hard

### Half Adder

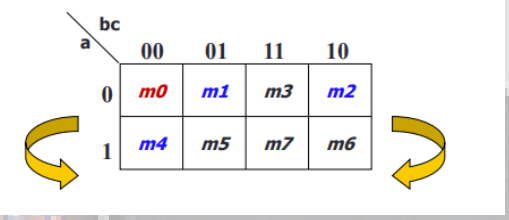
- Adds $(X,Y)$ producing $(C,S)$
- $C=X\cdot Y$, $S=X'\cdot Y+X\cdot Y'=X\oplus Y$

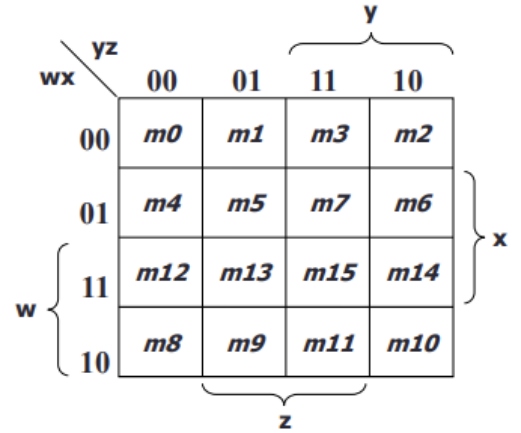



| Inputs | | Outputs | |
|---|---|---|---|
| X | Y | C | S |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

### K-maps

- Gray Code: Unweighted, only single bit change from one code value to next
- K-map: Systematic method to obtain simplified SOP expressions
- Each square is a minterm differing by one literal
- Has wrap-around
- Group as many cells as possible, select as few groups as possible to cover all the cells (minterms) of the function
- Implicant: Product term used to cover minterms of function
- Prime Implicant: Product term obtaned by combining the maximum possible number of minterms from adjcant squares in the map
- Essential Prime Implicants: Prime implicant that includes at least one minterm that is not covered by any other prime implicant
- SOP: Select minimum subset of prime implicants to cover minterms, and add them together
- Don't Care: Can be either 1 or 0, might help with simplifying k-maps
- POS: Draw k-map of complement, then apply DeMorgan's on the result

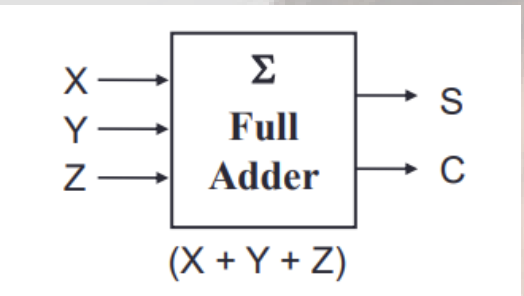


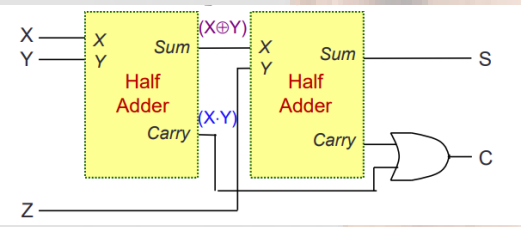



## 17. Combinational Circuits

### Full Adder

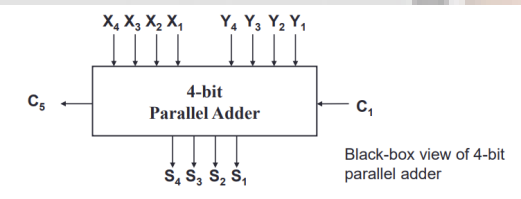
- To fully add two binary numbers, need 3 bits
- $C = X \cdot Y + X \cdot Z + Y \cdot Z$,
  $S = X' \cdot Y' \cdot Z + X' \cdot Y \cdot Z' + X \cdot Y' \cdot Z' + X \cdot Y \cdot Z$
- Can be made from 2 half adders and an OR gate
- Z is Cin, C is Cout

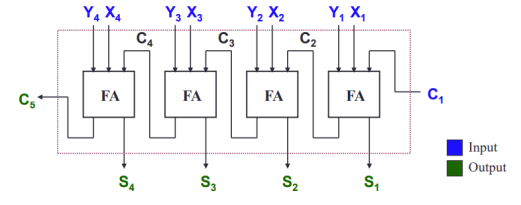



### 4-bit Parallel Adder

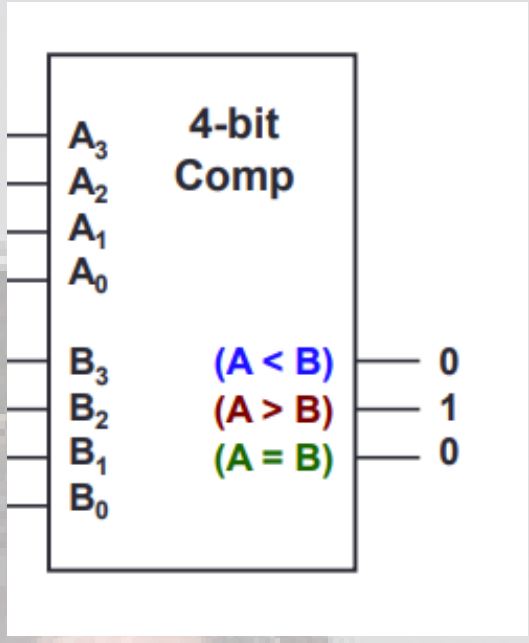
- Add two 4-bit numbers together with Cin to get 5-bit result
- Use addition formula for each pair of bits and Cin
- $C_{i+1}S_i = X_i + Y_i + C_i$
- $C_{i+1} = X_i \cdot Y_i + (X_i \oplus Y_i) \cdot C_i$, $S_i = X_i \oplus Y_i \oplus C_i$
- Can use same logic to expand to 16-bit parallel adder
- Example Application: Voting system




### Magnitude Comparator

- Check trichotomy



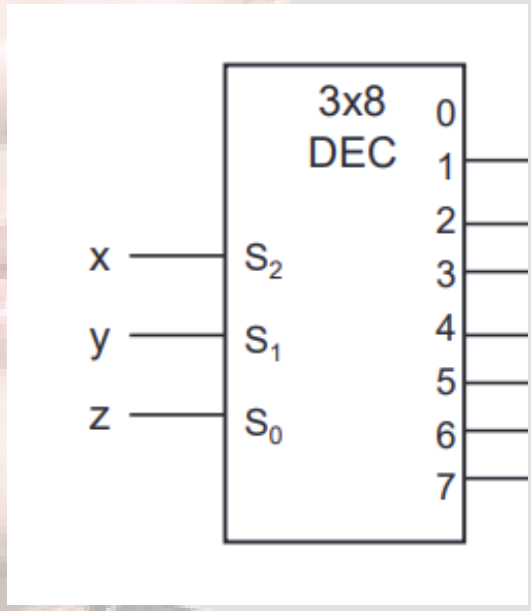

### Circuit Delays

- Max of each input / output
- Analyse components
- Full adder: If $C_i = mt$, $S_i = max(t, mt) + t$, $C_{i+1} = max(t, mt) + 2t$
- Parallel Adder: $S_n = (2(n-1) + 2)t$, $C_{n+1} = (2(n-1) + 3)t$

## 18. MSI Components

### Decoder

- Convert binary information from $n$ input lines to $2^n$ output lines, can be used to generate minterms
- Implemented with AND gates
- We can implement functions by using the output minterms
- Can have enable control signal, which is either one/zero-enable
- Can build larger decoders from smaller ones
- Can also have negated outputs

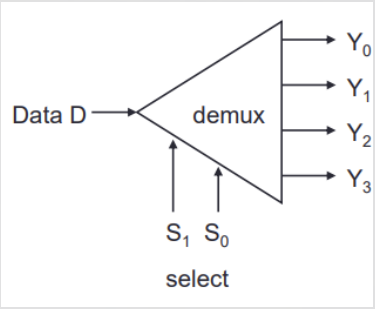



### Encoder

- Opposite of decoding
- Provides a code corresponding to input line
- Implemented with OR gates
- Priority Encoder: Input with highest priority takes precedence, all zero is invalid

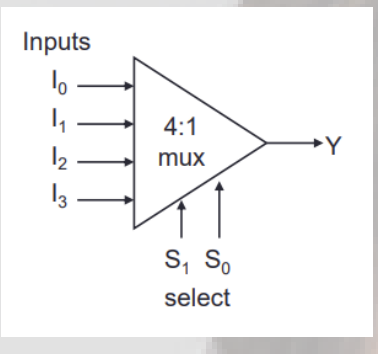
### Demultiplexer

- Directs data from input to one selected output line
- Identical to decoder with enable using data

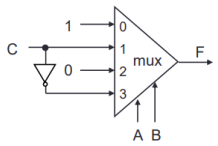



## Multiplexer

- Selects intput from one of the input lines and pipes it to output line
- Output is sum of (product of data lines and selection lines)
- Larger multiplexers can be constructed from smaller ones
- Constructing Function: Connect variables to selection lines, put 1 on data line if it is a minterm, 0 otherwise
- Squeezing Multiplexers: Use one variable less by moving it to the input data lines
- Group inputs by selector lines, and determine MUX input based on remaining variable and function




| A | B | C | F | MUX input |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 | |
| 0 | 1 | 0 | 0 | C |
| 0 | 1 | 1 | 1 | |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | |
| 1 | 1 | 0 | 1 | C' |
| 1 | 1 | 1 | 0 | |




## 19. Sequential Logic

- Syncronous: Outputs change at specific time, Asynchronous: Outputs change at any time
- Multivibrator: Class of sequential circuits, Bistable, Monostable, Astable
- Bistable logic devices: Latches and flip flops
- Memory Element: Device which can remember value indefinitely, or change value on command from inputs
- Clock: Usually a Square Wave
- Memory Element can be Pulse Triggered (Latches) or Edge triggered (Flip-flops)
- Flip-flops are synchronous bistable devices, and have a $>$ indicating clock input

## S-R Latch

- Inputs: $S, R$, Outputs: $Q, Q'$
- $Q$ high: SET state, $Q$ low: RESET state
- Gated: Has enable input, outputs can only change if enabled

| S | R | Q(t+1) | |
|---|---|---|---|
| 0 | 0 | Q(t) | No change |
| 0 | 1 | 0 | Reset |
| 1 | 0 | 1 | Set |
| 1 | 1 | indeterminate | |

$Q(t+1) = S + R' \cdot Q$

$S \cdot R = 0$

## Gated D Latch

- S-R latch, but $R = S'$
- Eliminates invalid state

| EN | D | Q(t+1) | |
|---|---|---|---|
| 1 | 0 | 0 | Reset |
| 1 | 1 | 1 | Set |
| 0 | X | Q(t) | No change |

When *EN*=1, **Q(t+1) = D**

## Asynchronous Inputs

- Affect state of flip-flop independent of clock
- Preset immediately sets output to HIGH
- Clear immediately sets output to LOW

## Flip-Flops

## Characteristic Table

| J | K | Q(t+1) | Comments |
|---|---|---|---|
| 0 | 0 | Q(t) | No change |
| 0 | 1 | 0 | Reset |
| 1 | 0 | 1 | Set |
| 1 | 1 | Q(t)' | Toggle |

| S | R | Q(t+1) | Comments |
|---|---|---|---|
| 0 | 0 | Q(t) | No change |
| 0 | 1 | 0 | Reset |
| 1 | 0 | 1 | Set |
| 1 | 1 | ? | Unpredictable |

| D | Q(t+1) | |
|---|---|---|
| 0 | 0 | Reset |
| 1 | 1 | Set |

| T | Q(t+1) | |
|---|---|---|
| 0 | Q(t) | No change |
| 1 | Q(t)' | Toggle |

## Excitation Table

| Q | $Q^+$ | J | K |
|---|---|---|---|
| 0 | 0 | 0 | X |
| 0 | 1 | 1 | X |
| 1 | 0 | X | 1 |
| 1 | 1 | X | 0 |

*JK* Flip-flop

| Q | $Q^+$ | S | R |
|---|---|---|---|
| 0 | 0 | 0 | X |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | X | 0 |

*SR* Flip-flop

| Q | $Q^+$ | D |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

*D* Flip-flop

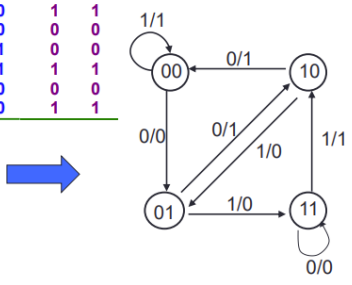
| Q | $Q^+$ | T |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

*T* Flip-flop

## State Table and State Diagram

- Use K-map to find flip-flop input functions based on current state and input
- Edge is input/output
- Nodes are states

| Present state | | Input | Next state | | Output | Flip-flop inputs | | | |
|---|---|---|---|---|---|---|---|---|---|
| A | B | x | $A^+$ | $B^+$ | y | JA | KA | JB | KB |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |

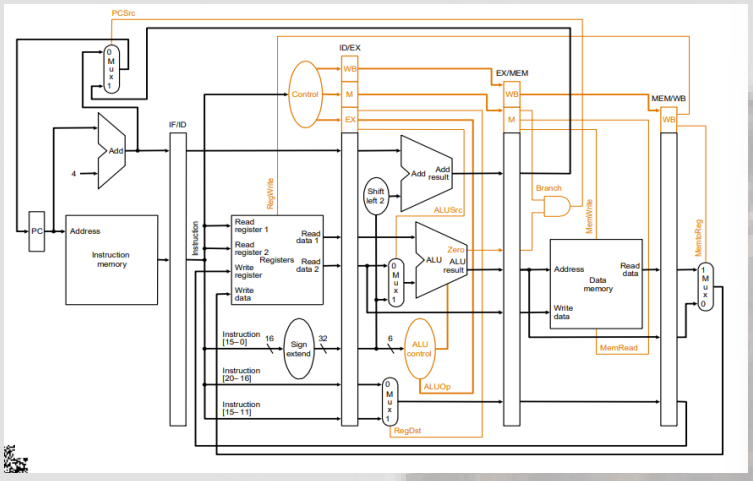



## Memory

- stores program and data
- 1 byte = 8 bits, 1 word is a multiple of bytes
- Hierarchy: Fast expensive volatile to Slow cheap non-volatile
- Memory unit stores information in words
- Has input (write) lines and output (read) lines
- Address consists of lines to choose location
- Static RAM: Uses flip-flops as memory cells, Dynamic RAM: Uses capacitor charges to represent data, have to be constantly refreshed
- Memory Array: Chips combined to form larger memory

## 20. Pipelining

- Does not help latency of single tasks, but rather throughput of entire workload
- Multipled tasks operating simultaneously using different resources
- Delays are due to slowest stage and stalling for dependencies
- Introduce Pipeline registers between stages




- Stages and information:
  - ID Stage: IF/ID supplies register numbers, offset, PC+4, ID/EX receives data values from register file, extended immediate, PC+4
  - EX Stage: ID/EX supplies above, EX/MEM receives (PC+4)+(Immd x 4), ALU result, isZero?, Data Read 2
  - MEM Stage: EX/MEM supplies above, MEM/WB receives ALU result, memory read data
  - WB Stage: MEM/WB supplies above, at the end, result is written back to register file
  - However, register provided by IF/ID is wrong, so we need to pass it from ID/EX all the way through
- Each control signal has an associated stage
- Single-Cycle Processor: Cycle Time = max(Total of time for each stage) for each instruction, Execution time for I instructions is multiplied
- Multi-Cycle Processor: Cycle Time = max(Time for a stage), Execution time for I instructions = I x Average Cycles per Instruction x Cycle Times
- Pipelining Processor: Cycle Time = max(Time for a stage) + Pipelining Overhead, Cycles for I instructions = I + N - 1, Execution time for I instructions = (I + N - 1) x Cycle Time
- Ideal Speedup: Each stage takes same amount of time, no pipelining overhead, number of instructions is much larger than number of stages, gives N times speedup

## 21. Pipeline Hazards

### Structural Hazard

- Simultaneous use of hardware resource
- e.g. If only single memory, memory could be accessed at the same time by 2 different instructions
- Stalling: Delay instruction for a cycle
- Separate Memory: Split memory into data and instruction memory
- Writing and Reading from memory at same time: Split cycle into half, first half for writing and second half for reading

### Instruction Dependencies

### Data Dependencies

- Overlap between data accessed by different instructions
- Read After Write / True data dependency: Later instruction reads from destination register written by earlier instruction
- Write after read, Write after write, do not cause pipeline hazards
- Result from execution is produced at end of EX stage, and only needed at start of EX stage of next instruction
- Solution: Forward the result to any later instructions, bypassing data read or ID
- However, this does not work for the result of LOAD instructions after MEM stage, so we still need stalling

### Control Dependencies

- Execution of instruction depends on another
- Control Dependency: Branch decision is made in MEM stage
- Stalling: Waiting until after MEM stage to fetch next instruction, introduces 3 cycle delay
- Techniques: Early branch resolution, Branch prediction, Delayed branching
- Early Branching: Make decision in ID stage instead of MEM, reduced to 1 cycle delay
- However, registers involved might be produced by instructions before, and further stalling might be needed
- Solve by adding forwarding path from ALU to ID stage, still has 1 cycle delay
- After a load instruction, early branching does not speedup
- Branch Prediction: Assume all branches are not taken
- If wrong guess is made, flush instructions
- Delayed Branching: Move non-control dependent instructions into slots following a branch (branch-delay slots)
- Executed regardless of outcome, if no such instruction, add no-op
- With early branching, we have 1 slot

### Summary

- Default: Wait till after MEM then get result in ID
- Forwarding: Forward from EX to next EX, or MEM to next EX
- Default: Wait till after MEM to get result to IF next instruction
- Early Branching: Compute result in ID, forward from previous EX
- Branch Prediction: Carry on and fetch next instruction, fetch correct instruction immediately after actual result is known
- List of delay durations:

  - Without data forwarding:

| Instruction | Delay Caused |
|---|---|
| RAW | +2 |
| Load Word | +2 |
| Branch at MEM | +3 |
| Branch at ID | +1 |

  - With data forwarding:

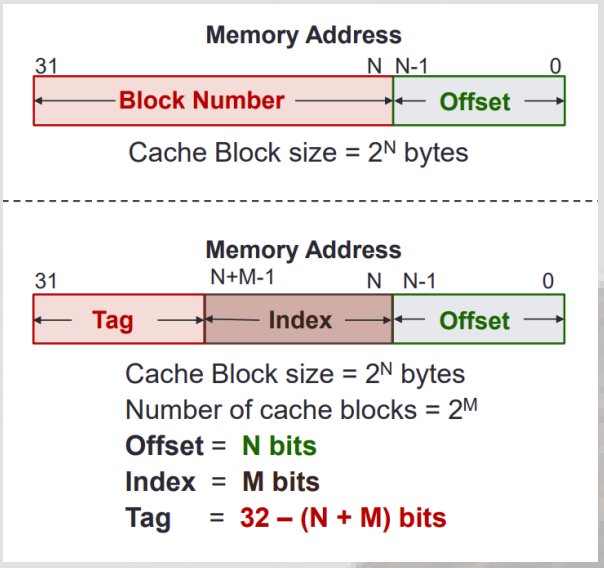
| Instruction | Delay Caused |
|---|---|
| RAW | +0 |
| Load Word | +1 |
| Branch at MEM | +3 |
| Branch at ID | +1 |
| RAW + Branch at ID | +1 |
| Load + Branch at ID | +2 |

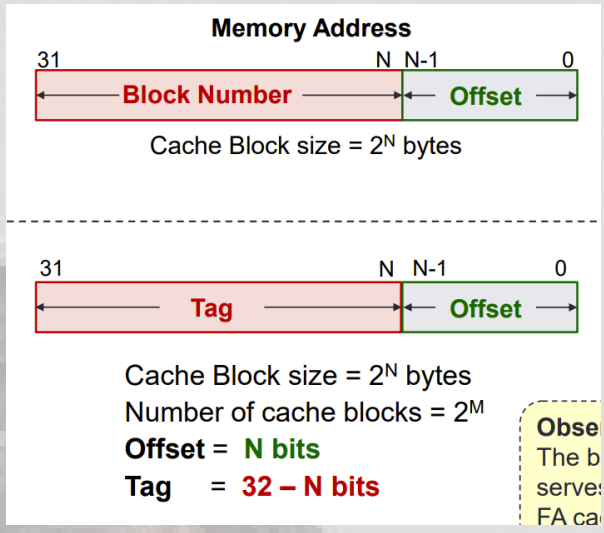
## 22. Cache

- Keep frequently and recently used data in smaller but faster memory
- Principle of Locality: Program access only a small portion of the memory address space within a small time interval
- Temporal Locality: Recently referenced items are likely to be referenced again soon
- Spatial Locality: Nearby items are likely to be referenced soon
- Cache Hit: Data is in cache, Rate: Fraction of memory access that hit, Time: Time to access cache
- Cache Miss: Data is not in cache, Rate: 1 - Hit rate, Penalty: Time to replace cache block + hit time
- Average Access Time = Hit Rate * Hit Time + (1 - Hit Rate) * Miss Penalty
- Cache Block/Line: Unit of transfer between memory and cache, typically one or more words
- MIPS: 1 word = 4 bytes = 32 bits
- Memory Address = (32-N) Block Number + N Offset

### Direct Mapped Cache

- Cache Index = Block Number mod Number of Cache Blocks
- If Number of Cache Blocks = $2^M$, then the last M bits are the cache index
- Multiple memory blocks can map to the same cache block when they have the same cache index
- They have different Tag = Block Number / Number of Cache Blocks
- If Cache Block Size = $2^N$ bytes and Number of Cache Blocks = $2^M$, then Memory Address = (32-N-M) bits Tag, M bits Index, N bits offset
- Block Number = Tag + Index bits
- Cache also has a valid bit indicating validity of data
- Example:
  - e.g. 4GB Memory, 16KB Cache, 16 byte blocks
  - Offset = $N = 2^4$ bits, Block Number = 32 - 4 = 28 bits
  - Number of Blocks = $2^{28}$
  - Number of Cache Blocks = 1024 = $2^{10}$
  - Cache Index = $M = 10$ bits
  - Cache Tag = 32 - 10 - 4 = 18 bits
- Offset = log2 Number of bytes in Block
- Number of Cache Blocks = Cache Size / Block Size
- Cache Index = log2 Number of Cache Blocks
- Tag = 32 - Offset - Cache Index
- Compulsory / Cold Miss: First access to block, needs to be brought into cache
- Conflict / Collision Miss: Several blocks are mapped to same block or set
- Capacity Miss: Blocks are discarded as cache cannot contain all blocks needed
- Cache and main memory could be inconsistent
- Write-through: Write data both to cache and to main memory
- Limited by memory speed, can use write buffer to speedup
- Write-back: Write data to cache, and only to main memory when evicted
- Wasteful if we write back every block, can use dirty bit to indicate whether data is changed
- On read miss, data loaded into cache and then loaded into register
- Write Allocate: Load complete block into cache, change only required word in cache, write to main memory depending on policy
- Write Around: Do not load block to cache, write directly to main memory only

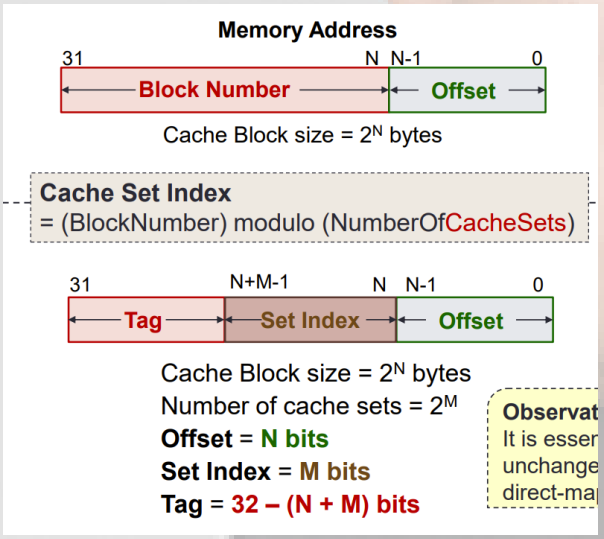



## 23. Set / Fully Associative Cache

### Set Associative Cache

- Block Size tradeoff: Large block size takes advantage of spatial locality, but larger miss penalty, and fewer blocks increases miss rate
- Set Associative Cache targets conflict misses
- N-way Set Associative Cache: Memory block can be in a fixed number of locations in cache
- Cache contains a number of sets, each with N blocks
- Memory block can be place in any of the N blocks after loading
- Cache Set Index = Block Number mod Number of Cache Sets
- Essentially the same as direct mapping bit and math wise
- Number of Sets = Cache Blocks / N-way
- Set Index = log2 Number of Sets
- Rule of Thumb: A direct mapped cache of size N has about the same miss rate as a 2-way set associative cache of size N/2




### Fully Associative Cache

- Memory block can be placed in any location in cache
- Cache Block size = $2^N$ bytes
- Number of Cache blocks = $2^M$
- Offset = N bits, Tag = 32 - N bits

### Performance / Replacement Policy

- Total Miss = Cold Miss + Conflict Miss + Capacity Miss
- Capacity Miss = Total Miss - Cold Miss when Conflict Miss = 0
- Conflict miss is 0 for fully associative cache
- Set Associative / Fully Associative Cache needs to replace blocks if full using block replacement policy
- Least Recently Used: Temporal Locality
- First in First Out, Random Replacement, Least Frequently Used