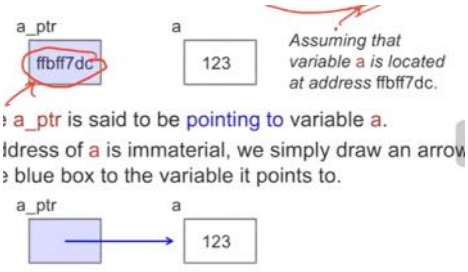
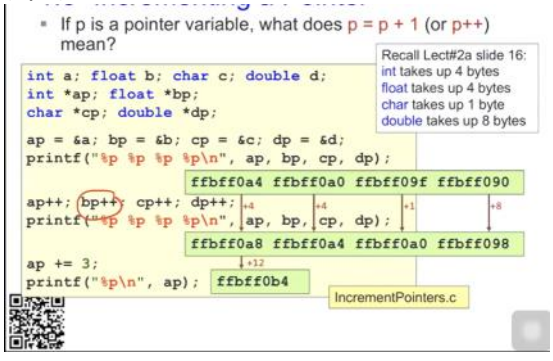
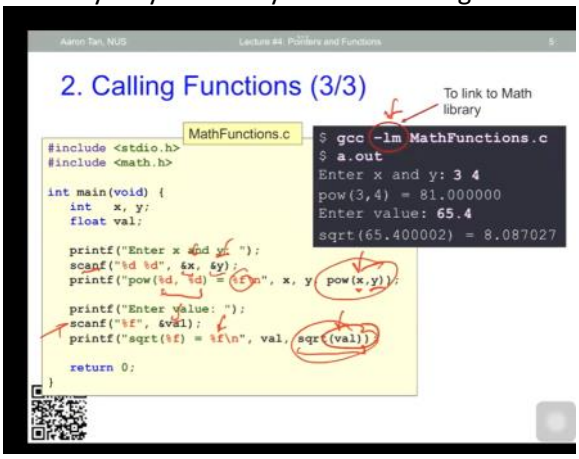
# Week 2

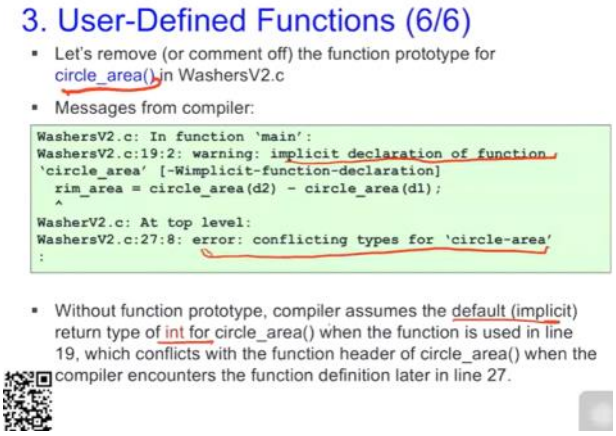
Monday, August 21, 2023 11:26 PM

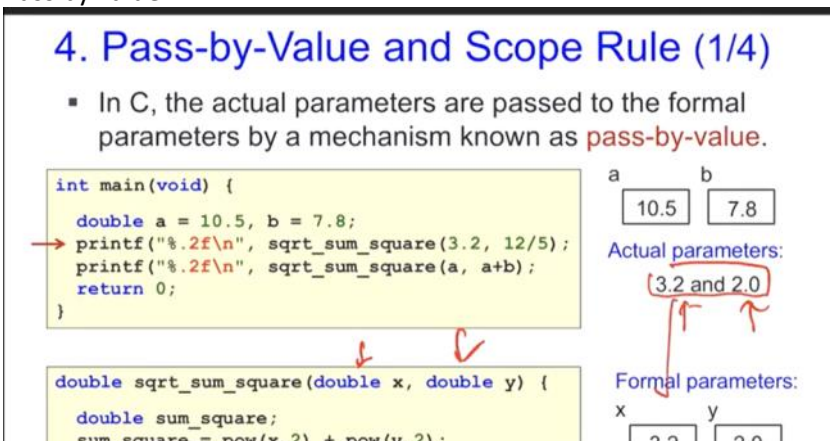
- A variable has name, data type, address
- Variable occupies space in memory so it needs address which system keeps track of
- 16 billion places
- Can get address using address operator & (use %p for pointer)
- Pointer is hexadecimal number
- Address of a variable varies from run to run, cuz system allocates any free memory to the var
- POINTER VARIABLE
  - Varibale that contains address of another variable

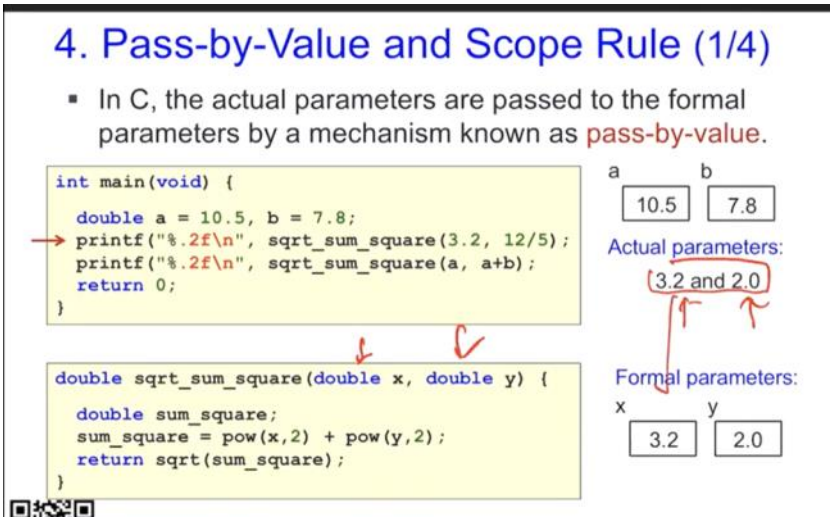


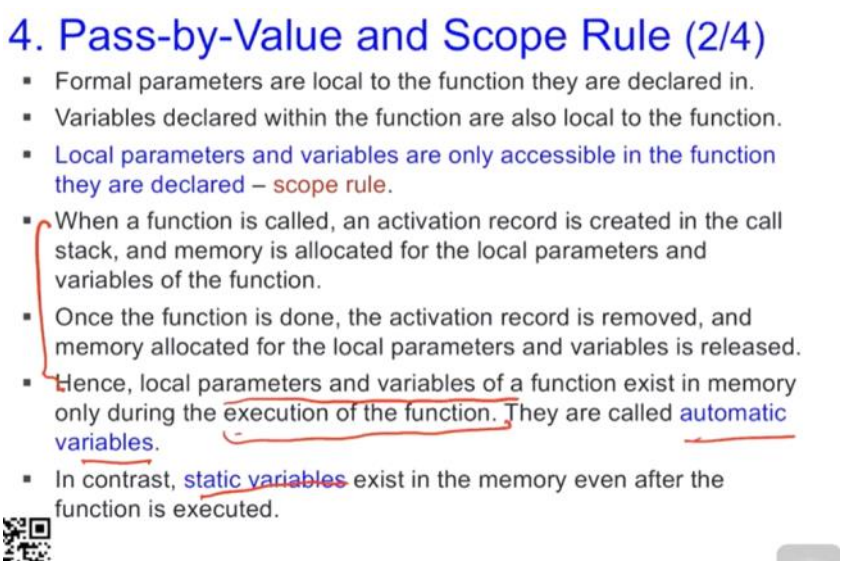

  - Declaring a pointer
  - type *pointer_name (readright to left)
  - Ex. int *a_ptr
  - Assigning value to pointer
  - Only an address may be assigned to pointer
  - Ex. a_ptr = &a;
  - All varible contain random data before initialization
  - Access variable through pointer
  - Can access "a" directly as usual or indirectly thorugh a_ptr by using the indirection operator also called dereferencing operator)*
  - `printf("a = %d\n", *a_ptr);`
  - *a_ptr = 456; changes value of a
- Incrementing pointer
  - (*p1)++
  - 

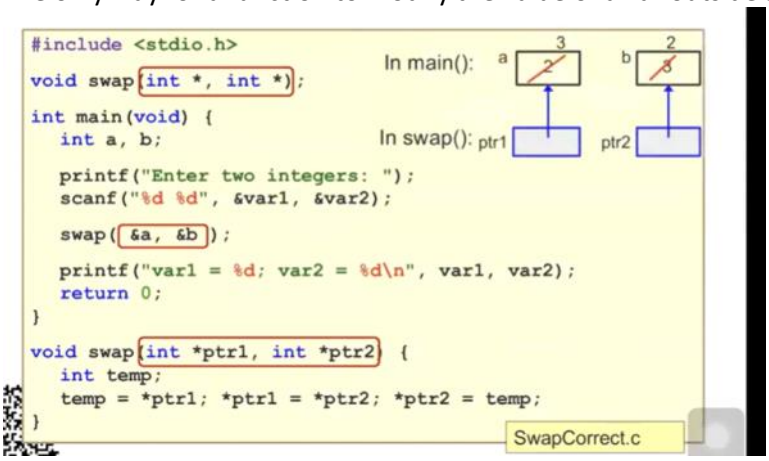

  - Cuz the n is allocated random numbers as it s not initialized
- Math library <math.h> AND + compile with -lm option
- The only way to modify variables is to give it the address - restriction of C
- 

- User-Defined Functions
- 

- Pass by value
- 

- 

- 

- The only way for a function to modify the value of a var outside scope is to use pointers
- 

- F
- ARRAYS

## 2.1 Array Declaration with Initializers

- As seen in ArraySumV2.c, an array can be initialized at the time of declaration.

```
// a[0]=54, a[1]=9, a[2]=10
int a[3] = {54, 9, 10};

// size of b is 3 with b[0]=1, b[1]=2, b[2]=3
int b[] = {1, 2, 3};

// c[0]=17, c[1]=3, c[2]=10, c[3]=0, c[4]=0
int c[5] = {17, 3, 10};
```

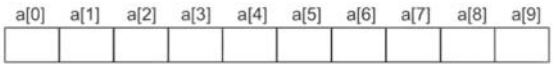
Note what happens when fewer initial values are provided.

- The following initializations are incorrect:

```
int e[2] = {1, 2, 3};  // warning issued: excess elements

int f[5];
f[5] = {8, 23, 12, -3, 6}; // too late to do this;
                           // compilation error
```

## 2.2 Arrays and Pointers

- Example: `int a[10]`




- When the array name `a` appears in an expression, it refers to the address of the first element (i.e. `&a[0]`) of that array.

```
int a[3];
printf("%p\n", a);
printf("%p\n", &a[0]);
printf("%p\n", &a[1]);
```

```
ffbff724
ffbff724
ffbff728
```

These 2 outputs wil always be the same.

Output varies from one run to another. Each element is of int type, hence takes up 4 bytes (32 bits).

- Cannot set an uninitialized array to equal another one
  - An array name is a fixed (constant) pointer ie it points to first element of array which cannot be changed.
  - You will be trying to alter the unintialized array to make It point elsewhere
  - Instead you must use a for loop to copy over
- In array prototype don't need to specify the size of array
- ** C ignores length information of array as parameter. So you need to add size as another param

## 2.4 Array Parameters in Functions (3/3)

- Since an array name is a pointer, the following shows the alternative syntax for array parameter in function prototype and function header in the function definition

```
int sumArray(int *, int); // fn prototype
```

```
// function definition
int sumArray(int *arr, int size) {
   ...
}
```

- Compare this with the [ ] notation

```
int sumArray(int [], int); // fn prototype
```

```
// function definition
int sumArray(int arr[], int size) {
   ...
```

- Since array name is pointer, no need to pass address to function
- A function can always modify the content of the array its received
- STRINGS
- A string is a array of char with a null character '\0' at the end of the array
- Then you can use string funcitons by importing <string.h>
- Declare: char str[6];
- Assign: str[0] = 'e'; str[3] = '\0'

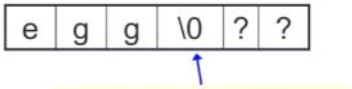
## 3.1 Strings: Basic

- Declaration of an array of characters

```
char str[6];
```

- Assigning character to an element of an array of characters

```
str[0] = 'e';
str[1] = 'g';
str[2] = 'g';
str[3] = '\0';
```




Without '\0', it is just an array of character, not a string.

Do not need '\0' as it is automatically added.

- Initializer for string
  - Two ways:

```
char fruit_name[] = "apple";
char fruit_name[] = {'a','p','p','l','e','\0'};
```

- Single quote is for 1 character
- Double quote produces a string followed by null terminator

## 3.2 Strings: I/O (1/3)

- Read string from stdin (keyboard)

```
fgets(str, size, stdin) // reads size - 1 char,
                        // or until newline
scanf("%s", str);  // reads until white space
```

- Print string to stdout (monitor)

```
puts(str);  // terminates with newline
printf("%s\n", str);
```

Note: There is another function gets(str) to read a string interactively. However, due to security reason, we avoid it and use fgets() function instead.

- scanf reads until first white space
- fgets reads until newline

## 3.2 Strings: I/O (3/3)

StringIO1.c

```
#include <stdio.h>
#define LENGTH 10

int main(void) {
   char str[LENGTH];

   printf("Enter string (at most %d characters): ", LENGTH-1);
   scanf("%s", str);
   printf("str = %s\n", str);
   return 0;
}
```

Test out the programs with this input:
My book

Output:
str = My

StringIO2.c

```
#include <stdio.h>
#define LENGTH 10

int main(void) {
   char str[LENGTH];

   printf("Enter string (at most %d characters): ", LENGTH-1);
   fgets(str, LENGTH, stdin);
   printf("str = ");
   puts(str);
   return 0;
}
```

Output:
str = My book

Note that puts(str) adds a newline automatically.

- By changing the last part of the string to null, the length is changed

RemoveVowels.c

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
int main(void) {
   int i, len, count = 0;
   char str[101], newstr[101];

   printf("Enter a string (at most 100 characters): ");
   fgets(str, 101, stdin); //what happens if you use scanf() here?
   len = strlen(str); // strlen() returns number of char in string
   if (str[len - 1] == '\n')
      str[len - 1] = '\0';
   len = strlen(str); // check length again

   for (i=0; i<len; i++) {
      switch (toupper(str[i])) {
         case 'A': case 'E':
         case 'I': case 'O': case 'U': break;
         default: newstr[count++] = str[i];
      }
   }
   newstr[count] = '\0';
   printf("New string: %s\n", newstr);
   return 0;
}
```

## 3.4 String Functions (1/2)

- C provides a library of string functions
  - Must include <string.h>
  - Here are a few commonly used string functions
- strlen(s)
  - Return the number of characters in s
- strcmp(s1, s2)
  - Compare the ASCII values of the corresponding characters in strings s1 and s2.
  - Return
    - a negative integer if s1 is lexicographically less than s2, or
    - a positive integer if s1 is lexicographically greater than s2, or
    - 0 if s1 and s2 are equal.
- strncmp(s1, s2, n)
  - Compare first n characters of s1 and s2.

## 3.4 String Functions (2/2)

- strcpy(s1, s2)
  - Copy the string pointed to by s2 into array pointed to by s1.
  - Function returns s1.
  - Example:

```
char name[10];
strcpy(name, "Matthew");
```

| M | a | t | t | h | e | w | \0 | ? | ? |
|---|---|---|---|---|---|---|---|---|---|

  - The following assignment statement does not work:

```
name = "Matthew";
```

  - What happens when string to be copied is too long?

```
strcpy(name, "A very long name");
```

| A | | v | e | r | y | | l | o | n | g | | n | a | m | e | \0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

- Don't use strcpy
- Use strncpy(s1, s2, n) copys at most n characters

## 3.5 Importance of '\0' in a String (1/2)

- To be treated as a string, the array of characters must be terminated with the null character '\0'.
- Otherwise, string functions will not work properly on it.
- For instance, the printf("%s", str) statement will print until it encounters a null character in str.
- Likewise, strlen(str) will count the number of characters up to (but not including) the null character.
- In many cases, a string that is not properly terminated with '\0' will result in illegal access of memory.

## 3.5 Importance of '\0' in a String (2/2)

- What is the output of this code?

WithoutNullChar.c

```
#include <stdio.h>
#include <string.h>

int main(void) {
  char str[10];

  str[0] = 'a';
  str[1] = 'p';
  str[2] = 'p';
  str[3] = 'l';
  str[4] = 'e';

  printf("Length = %d\n", strlen(str));
  printf("str = %s\n", str);

  return 0;
}
```

One possible output:

```
Length = 5
str = apple
```

Compare the output if you add:

```
str[5] = '\0';
```

or, you have:

```
char str[10] = "apple";
```

printf() will print %s from the starting address of str until it encounters the '\0' character.

- STRUCTURES
- A "group" can be a member of another "group"
- Structure type
  - "typedef struct" is a keyword
  - Then have elements in curly braces

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

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

```
typedef struct {
   int stuNum;
   float score;
   char grade;
} result_t;
```

  - Creates a new data type ex. Account_t
  - A type is not a variable
  - No memory is allocated to a type
  - Struct is only definition of type
- Structure variables

```
typedef struct {
   int stuNum;
   float score;
   char grade;
} result_t;

result_t result1 = { 123321, 93.5, 'A' };
```

```
typedef struct {
   int day, month, year;
} date_t;

typedef struct {
   int cardNum;
   date_t expiryDate;
} card_t;

card_t card1 = {888888, {31, 12, 2020}};
```

- Also can be done using dot operator

```
result_t result2;

result2.stuNum = 456654;
result2.score = 62.0;
result2.grade = 'D';
```

- 
```
card_t card2 = { 666666, {30, 6} };

card2.expiryDate.year = 2021;
```

- Use dot notation to print out

## 4.6 Reading a Structure Member

- The structure members are read in individually the same way as we do for ordinary variables
- Example:

- 
```
result_t result1;

printf("Enter student number, score and grade: ");
scanf("%d %f %c", &result1.stuNum, &result1.score,
                  &result1.grade);
```

- Must provide address because we want scan to be able to modify
- Assigning structures
  - Can use the dot operator to access indiv. Member of a structure variable
  - If structure variable's name is used, we are referring to the entire structure
  - Asssignments can be done to structures
  - Result2 = result1;

```
result2 = result1;
```
=
```
result2.stuNum = result1.stuNum;
result2.score = result1.score;
result2.grade = result1.grade;
```

- *Before:*

result1

| stuNum | score | grade |
|---|---|---|
| 123321 | 93.5 | 'A' |

result2

| stuNum | score | grade |
|---|---|---|
| 456654 | 62.0 | 'D' |

## 4.8 Returning Structure from Function (1/3)

- Example:
  - Given this structure type result_t,

```
typedef struct {
    int max;
    float ave;
} result_t;
```

- 
  - Define a function func() that returns a structure of this type:

```
result_t func( ... ) {

}
```

  - To call this function:

```
result_t result;

result = func( ... );
```

## 4.8 Returning Structure from Function (2/3)

StructureEg2.c

```
#include <stdio.h>

typedef struct {
    int max;
    float ave;
} result_t;

result_t max_and_average(int, int, int);

int main(void) {
    int num1, num2, num3;
    result_t result;

    printf("Enter 3 integers: ");
    scanf("%d %d %d", &num1, &num2, &num3);
    result = max_and_average(num1, num2, num3);

    printf("Maximum = %d\n", result.max);
    printf("Average = %.2f\n", result.ave);
    return 0;
}
...
```

StructureEg2.c

```
// Computes the maximum and average of 3 integers
result_t max_and_average(int n1, int n2, int n3) {
    result_t result;

    result.max = n1;
    if (n2 > result.max)
      result.max = n2;
    if (n3 > result.max)
      result.max = n3;

    result.ave = (n1+n2+n3)/3.0;

    return result;
}
```

- POINTERS TO STRUCTURES
- When you pass a structure in a fucntion it's the same as assigning the structure to the param
- Which means the entire structure is copied. The members of the actual param is copied into the corresponding members of the formal param
- Pass by value

## 4.9 Passing Structure to Function (2/2)

PassStructureToFn.c

```
// #include statements and definition
// of player_t are omitted here for brevity
void print_player(char [], player_t);

int main(void) {
    player_t player1 = { "Brusco", 23, 'M' }, player2;

    strcpy(player2.name, "July");
    player2.age = 21;
    player2.gender = 'F';

    print_player("player1", player1);
    print_player("player2", player2);

    return 0;
}

// Print player's information
void print_player(char header[], player_t player) {
    printf("%s: name = %s; age = %d; gender = %c\n", header,
           player.name, player.age, player.gender);
}
```

Passing a structure to a function

Receiving a structure from the caller

- Ex. NearbyStores.c
- Combining structures and array is good

PassStructureToFn2.c

```
// #include statements, definition of player_t,
// and function prototypes are omitted here for brevity
int main(void) {
   player_t player1 = { "Brusco", 23, 'M' };

   change_name_and_age(player1);
   print_player("player1", player1);
   return 0;
}

// To change a player's name and age
void change_name_and_age(player_t player) {
   strcpy(player.name, "Alexandra");
   player.age = 25;
}

// Print player's information
void print_player(char header[], player_t player) {
   printf("%s: name = %s; age = %d; gender = %c\n", header,
          player.name, player.age, player.gender);
}
```

player1: name = Brusco; age = 23; gender = M

- Value of player1 isn't changed because pass by value.
- To allow the function to modify content of the original structure variable can be done but you need to pass in the address of it
- PassAddrStructToFn.c

```
// #include statements, definition of player_t,
// and function prototypes are omitted here for brevity
int main(void) {
   player_t player1 = { "Brusco", 23, 'M' };

   change_name_and_age(&player1);
   print_player("player1", player1);
   return 0;
}

// To change a player's name and age
void change_name_and_age(player_t *player_ptr) {
   strcpy((*player_ptr).name, "Alexandra");
   (*player_ptr).age = 25;
}

// Print player's information
void print_player(char header[], player_t player) {
   printf("%s: name = %s; age = %d; gender = %c\n", header,
          player.name, player.age, player.gender);
}
```

- Dot operator has higher precedence than * dereference
- So you need the parenthesis
- The arrow operator (->)
  - Shortcut
  - `(*player_ptr).name` *is equivalent to* `player_ptr->name`
  - `(*player_ptr).age` *is equivalent to* `player_ptr->age`

# tut01qns

Thursday, August 24, 2023 10:16 PM

tut01qns

**CS2100 Computer Organization**
**Tutorial 1: C and Number Systems**
**(Week 3: 28 Aug – 1 Sep 2023)**

**Note: Friday's tutorial groups will have tutorial 1 on 25 August in lieu of 1 September. Refer to Canvas announcement for more information.**

1. In 2's complement representation, "sign extension" is used when we want to represent an *n-bit* signed integer as an *m-bit* signed integer, where $m > n$. We do this by copying the MSB (most significant bit) of the *n-bit* number $m - n$ times to the left of the *n-bit* number to create the *m*-bit number.

   For example, we want to sign-extend 0b0110 to an 8-bit number. Here $n = 4$, $m = 8$, and thus we copy the MSB bit 0 four (8 – 4) times, giving 0b00000110.

   Similarly, if we want to sign-extend 0b1010 to an 8-bit number, we would get 0b11111010.

   Show that IN GENERAL, sign extension is value-preserving. For example, 0b00000110 = 0b0110 and 0b11111010 = 0b1010.

2. We generalize $(r - 1)$'s-complement (also called *radix diminished complement*) to include fraction as follows:

   $$\textbf{(}r - 1\textbf{)'s complement of } N = r^n - r^{-m} - N$$

   where $n$ is the number of integer digits and $m$ the number of fractional digits. (If there are no fractional digits, then $m = 0$ and the formula becomes $r^n - 1 - N$ as given in class.)

   For example, the 1's complement of 011.01 is $(2^3 - 2^{-2}) - 011.01 = (1000 - 0.01) - 011.01 = 111.11 - 011.01 = 100.10$. (Since 011.01 represents the decimal value 3.25 in 1's complement, this means that -3.25 is represented as 100.10 in 1's complement.)

   Perform the following binary subtractions of values represented in 1's complement representation by using addition instead. (Note: Recall that when dealing with complement representations, the two operands must have the same number of digits.)

   (a) 0101.11 – 010.0101
   (b) 010111.101 – 0111010.11

   Is sign extension used in your working? If so, highlight it.

   Check your answers by converting the operands and answers to their actual decimal values.

3. Convert the following numbers to fixed-point binary in 2's complement, with 4 bits for the integer portion and 3 bits for the fraction portion.

   (a) 1.75 (b) -2.5 (c) 3.876 (d) 2.1

   Using the binary representations you have derived, convert them back into decimal. Comment on the compromise between range and accuracy of the fixed-point binary system.

4. [AY2010/2011 Semester 2 Term Test #1]
How would you represent the decimal value –0.078125 in the IEEE 754 single-precision representation? Express your answer in hexadecimal. Show your working.

5. Given the partial C program shown below, complete the two functions: **readArray()** to read data into an integer array (with at most 10 elements) and **reverseArray()** to reverse the array. For **reverseArray()**, you are to provide two versions: an iterative version and a recursive version. For the recursive version, you may write an auxiliary/driver function to call the recursive function.

```
#include <stdio.h>
#define MAX 10

int readArray(int [], int);
void printArray(int [], int);
void reverseArray(int [], int);

int main(void) {
   int array[MAX], numElements;

   numElements = readArray(array, MAX);
   reverseArray(array, numElements);
   printArray(array, numElements);

   return 0;
}

int readArray(int arr[], int limit) {

   // ...
   printf("Enter up to %d integers, terminating with a negative
integer.\n", limit);
   // ...
}

void reverseArray(int arr[], int size) {

   // ...
}

void printArray(int arr[], int size) {
   int i;

   for (i=0; i<size; i++) {
      printf("%d ", arr[i]);
   }
   printf("\n");
}
```

6. Trace the following program manually (do not run it on a computer) and write out its output. When you present your solution, draw diagrams to explain.

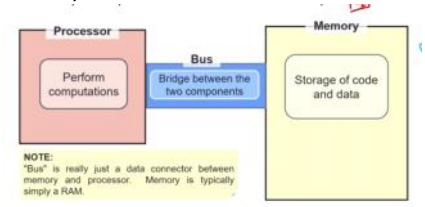
```
#include <stdio.h>

int main(void) {
   int a = 3, *b, c, *d, e, *f;

   b = &a;
   *b = 5;
   c = *b * 3;
   d = b;
   e = *b + c;
   *d = c + e;
   f = &e;
   a = *f + *b;
   *f = *d - *b;

   printf("a = %d, c = %d, e = %d\n", a, c, e);
   printf("*b = %d, *d = %d, *f = %d\n", *b, *d, *f);

   return 0;
}
```

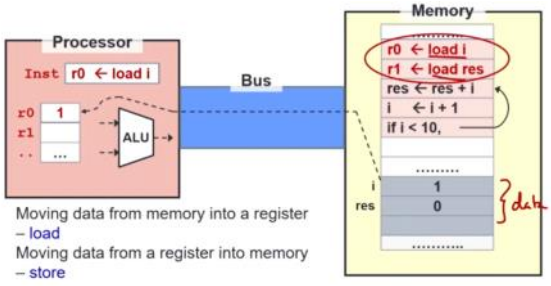
Remember to post on the Canvas forum or QnA if you have any queries.

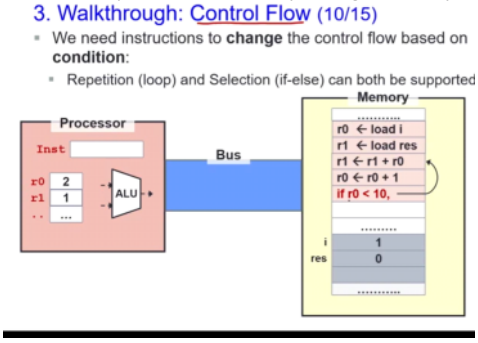
# Week 3

Friday, September 1, 2023 11:01 PM

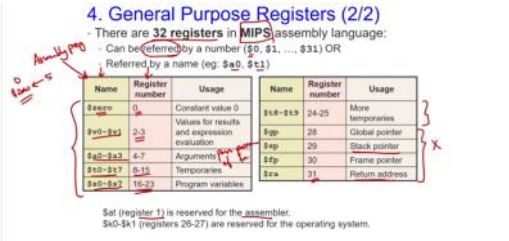
MIPS - microprocessor without interlocked pipelined stages

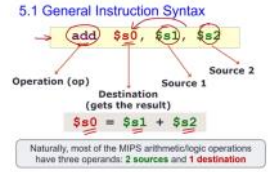
- Instruction Set Architecture
  - Abstraction on the interface between hardware and low level software
  - ASSEMBLER coverts from Assembly to Machine Code
- Components
  - Processor - performs computations
  - Memory - store code and data
  - 

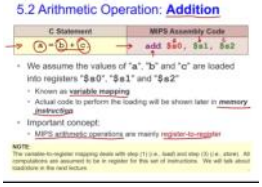
  - Bus is the wires and graits that connect
  - Memory is actually much slower than processor. So most of the time would have been taken for processor waiting for values from memory
  - So to avoid frequent access of memory,
    - We provide temporary storage for values in the processor known as REGISTERS
    - Registers works at speed of processor
  - We need instructions to move data into registers - instruction called LOAD
  - 

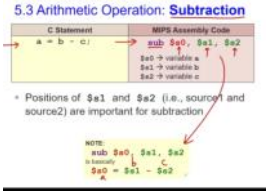
  - Arithmetic ops can work directly on registers only
  - 

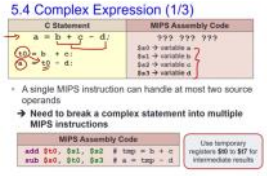
  - Both instruction and data are stored in memory
  - Load store model
    - Limits memory ops and relies on register for storage during execution
    - Major types of assembly instruction
      - Memory - move vals between mem and register
      - Calculation - arithmetic other ops
      - Control flow - change the sequential execution
- General Purpose Registers
  - Fast memories in processor
    - Data is transferred from mem to register for faster processing
  - Limited
    - Typically 16-32 register
    - Compiler associated var in program with registers
  - Registers don't have data type
    - Machine/assemble instruction assumes data stored in register is of correct type
  - Add - 2's/ addu - binary / fadd - floating
- MIPS has 32 register ($0, $1 …$31)

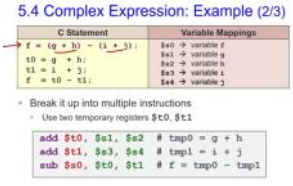
- 

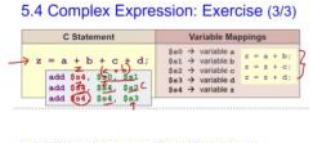

- MIPS assembly language
  - Each instruction executes a simple cmd
  - Each ln of assemble code contains at most 1 instruction
  - # is a comment
  - 

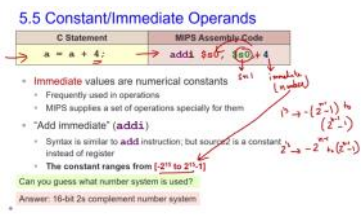
  - Variable mapping - mapping the values to registers in mips processor
  - 

  - 

  - 

  - 

  - 

  - 

  - There is no subi. Just use addi with negative constant
- Register Zero $0

- 5.6 Register Zero ($0 or $zero)
  - The number zero (0), appears very often in code
    - Provide register zero ($0 or $zero) which always have the value 0

| C Statement | MIPS Assembly Code |
|---|---|
| f = g; | add $s0, $s1, $zero |
| f = g + 0 | $s0 → variable f; $s1 → variable g; $zero → 0 |

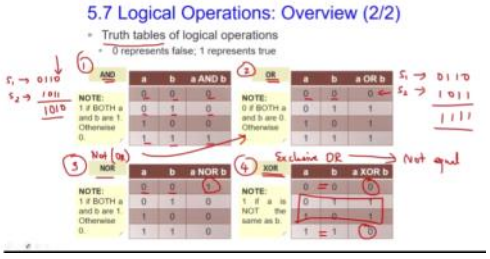
  - The above assignment is so common that MIPS has an equivalent **pseudo-instruction (move)**:

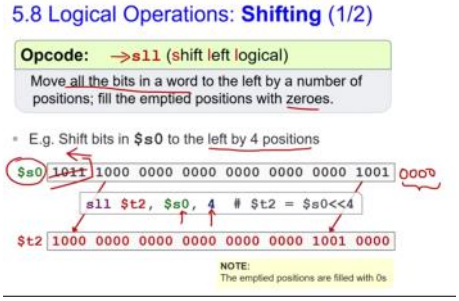
| MIPS Assembly Code | Pseudo-Instruction |
|---|---|
| move $s0, $s1 | "Fake" instruction that gets translated to corresponding MIPS instruction(s). Provided for convenience in coding only. |

LOGICAL OPERATIONS

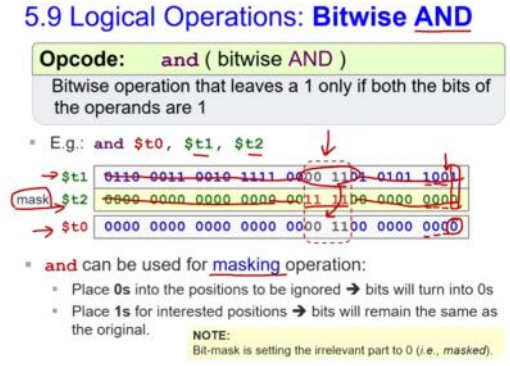
- Arithmetic instructions view content of register as single quantity
- We can view register as 32 raw bits rather than 32 bit number
- 

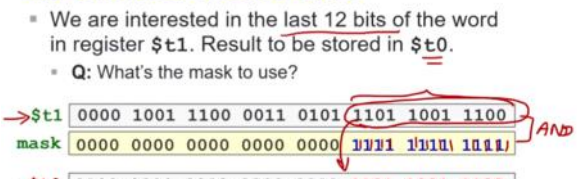
| Logical operation | C operator | Java operator | MIPS instruction |
|---|---|---|---|
| Shift Left | << | << | sll |
| Shift right | >>** | >>, >>> | srl |
| Bitwise AND | & | & | and, andi |
| Bitwise OR | \| | \| | or, ori |
| Bitwise NOT* | ~ | ~ | nor |
| Bitwise XOR | ^ | ^ | xor |

- 


- Max shifts is 31
- Shifting left is same as multiply by 2^n
- Shifting right is same as divide by 2^n
- Shifting Is much faster than multiplication/division

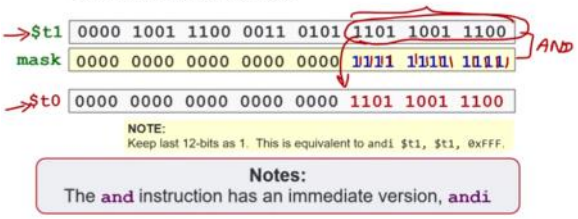



If you want specific bits of a word, use a AND mask with 1s in the places u want

## 5.9 Exercise: **Bitwise AND**

- We are interested in the last 12 bits of the word in register `$t1`. Result to be stored in `$t0`.
  - **Q:** What's the mask to use?




NOTE:
Keep last 12-bits as 1. This is equivalent to `andi $t1, $t1, 0xFFF`.

**Notes:**
The `and` instruction has an immediate version, `andi`

If you want specific bits of a word, use a AND mask with 1s in the places u want

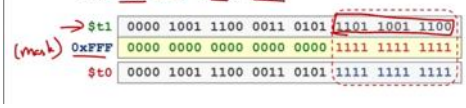
## 5.10 Logical Operations: **Bitwise OR**

**Opcode:** `or` ( bitwise OR )

Bitwise operation that places a 1 in the result if either operand bit is 1

**Example:** `or $t0, $t1, $t2`

- The `or` instruction has an immediate version `ori`
- Can be used to force certain bits to 1s
- E.g.: `ori $t0, $t1, 0xFFF`




Use or mask with 1s where you want to force a 1, and 0 where you keep original val

- NOR operation can be used to get NOT instruction

## 5.11 Logical Operations: **Bitwise NOR**

- Strange fact 1:
  - There is no **NOT** instruction in MIPS to toggle the bits (1 → 0, 0 → 1)
  - However, a **NOR** instruction is provided:

**Opcode:** `nor` ( bitwise NOR )

**Example:** `nor $t0, $t1, $t2`


- Question: How do we get a NOT operation?
- Question: Why do you think is the reason for not providing a NOT instruction?

To get not, do nor operation with zeros

- Nor $t0, $t0, $zero. Do Nor op with zeros to get NOT
- No NOT op because want to keep instruction set small

- Can use XOR op to get NOT
  - By pairing with all 1

## 5.12 Logical Operations: **Bitwise XOR**

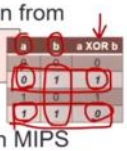
**Opcode:** `xor` ( bitwise XOR )

**Example:** `xor $t0, $t1, $t2`

- Question: Can we also get **NOT** operation from XOR?

Yes, let `$t2` contain all 1s.

```
xor $t0, $t0, $t2
```




- Strange Fact 2:
  - There is no **NORI**, but there is **XORI** in MIPS
  - Why?

NOTE:
A possible reason is that there is not much need for NORI. So there is no reason to add this capability to keep the processor design simple.

To get not, can also do xor op with 1s

- Addi is for 16-bit 2s comple
- So for large constants? 32 bit constant?
- Use "load upper immediate" (lui) to set upper 16 bit
- Use "or immediate" (ori) to set lower order bits

addi → 16-bit 2's comp → $-2^{15}$ to $(2^{15}-1)$

## 6. Large Constant: Case Study

- Question: How to load a 32-bit constant into a register? e.g 10101010 10101010 11110000 11110000

1. Use "load upper immediate" (lui) to set the upper 16-bit:

```
lui  $t0, 0xAAAA      #1010101010101010
```

| 1010101010101010 | 0000000000000000 |
|---|---|

Lower-order bits filled with zeros.

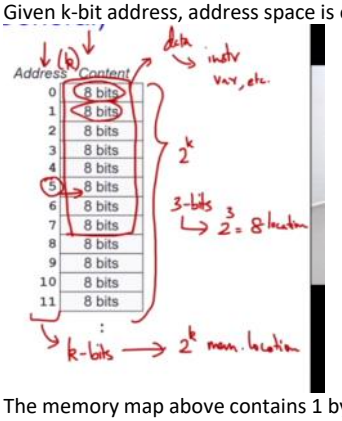
2. Use "or immediate" (ori) to set the lower-order bits:

```
ori  $t0, $t0, 0xF0F0 #1111000011110000
```

Higher-order bits filled with zeros.

| | | |
|---|---|---|
| | 1010101010101010 | 0000000000000000 |
| ori | 0000000000000000 | 1111000011110000 |

Memory (RAM) Organization

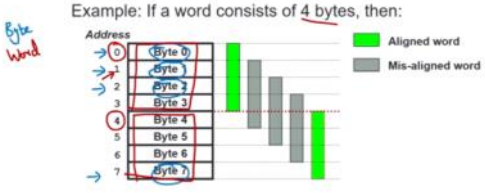
- Can be viewed as a single dimension arr of memory locations
  - Each location of memory has address which is index into arr
    - Given k-bit address, address space is of size 2^k
    - 

    - The memory map above contains 1 byte in every location - this is byte addressing
  - Memory -- transfer unit
    - With distinct memory address, can access
      - Single byte
      - Or single word (word addressable)
    - Word is
      - 2^n bytes
      - Common unit of tranfer betwee processor and memory
      - Commonly coincide with register size, integer and instruction size

### 1.2 Memory: Word Alignment

- **Word alignment**:
  - Words are aligned in memory if they begin at a byte address that is a multiple of the number of bytes in a word.

Example: If a word consists of 4 bytes, then:

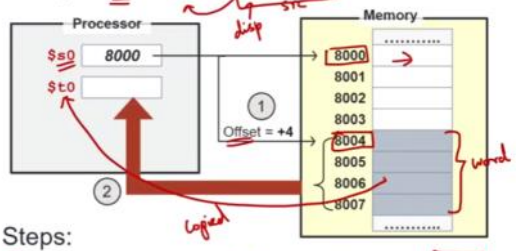



    - To quickly check if a given memory address is word aligned? Do address/4 - which is word size
  - MIPS memory instructions
    - MIPS is load-store register archi
    - 32 register - each 32 bit long
    - Each word contians 32 bits
    - Memory addresses are 32 bit long

| Name | Examples | Comments |
|---|---|---|
| 32 registers | $s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at | Fast processor storage for data. In MIPS, data must be in registers to perform arithmetic. |
| $2^{30}$ memory words | Mem[0], Mem[4], ..., Mem[4294967292] | Accessed only by data transfer instructions. MIPS uses **byte addresses**, so consecutive words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls. |

## 2.1 Memory Instruction: **Load Word**

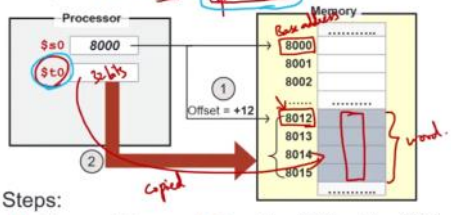
- Example: `lw $t0, 4($s0)`




- Steps:
  1. Memory Address = $s0 + 4 = 8000 + 4 = **8004**
  2. Memory word at `Mem[8004]` is loaded into $t0

## 2.2 Memory Instruction: **Store Word**

- Example: `sw $t0, 12($s0)`




- Steps:
  1. Memory Address = $s0 + 12 = 8000 + 12 = **8012**
  2. Content of $t0 is stored into word at `Mem[8012]`

- Only load and store instructions can access data in memory

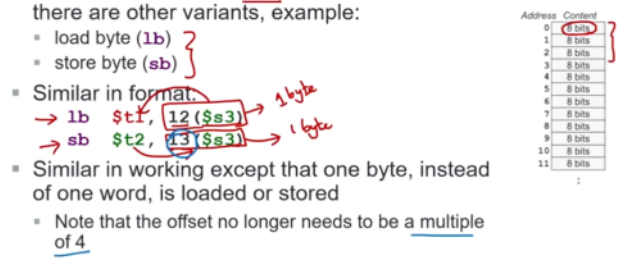
## 2.3 **Load** and **Store** Instructions

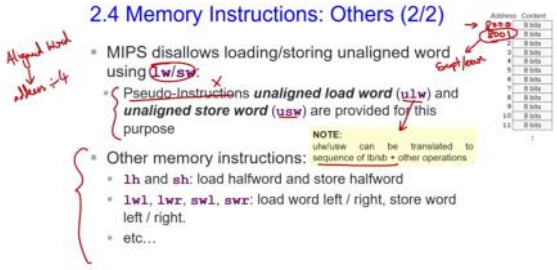
- Only `load` and `store` instructions can access data in memory.
- Example: Each array element occupies a word.

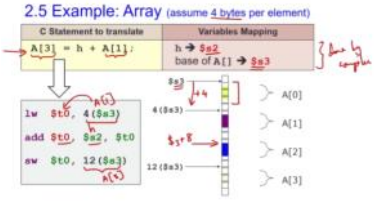
| C Code | MIPS Code |
|---|---|
| A[7] = h + A[10] | lw $t0, 40($s3) |
| | add $t0, $s2, $t0 |
| | sw $t0, 28($s3) |

- Each array element occupies a word (4 bytes).
- $s3 contains the **base address** (address of first element, A[0]) of array A. Variable h is mapped to $s2.
- Remember arithmetic operands (for `add`) are registers, not memory!

- Load/store word would load 4 bytes. Load/store byte would get 1 byte

- 

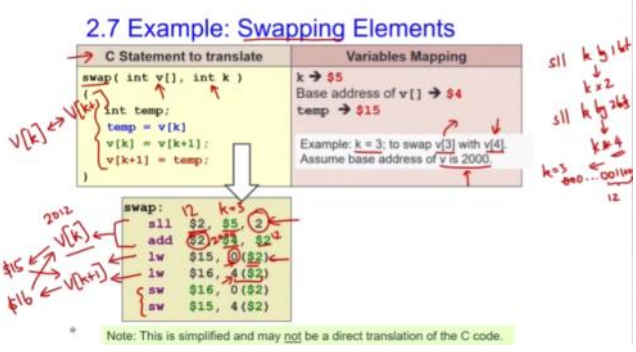

- 


- 


---

- Registers don't have types
- Register can hold any 32 bit number
  - The number has no implicit data type and its interpreted according to the instruction that uses it

SWAPPING ELEMENTS

- 

## 3. Making Decisions (2/2)

- Decision-making instructions
  - Alter the control flow of the program
  - Change the next instruction to be executed
- Two types of decision-making statements in MIPS

1. **Conditional** (branch)

```
bne $t0, $t1, label
beq $t0, $t1, label
```

2. **Unconditional** (jump)

```
j label
```

NOTE:
**Selection:** branch/jump down
**Repetition:** branch/jump up

In both cases, it's a "goto" operation.

- A label is an "anchor" in the assembly code to indicate point of interest, usually as branch target
  - Labels are NOT instructions!

## 3.1 Conditional Branch: **beq** and **bne**

- Processor follows the branch only when the condition is satisfied (true)
- **beq $r1, $r2, L1**
  - Go to statement labeled `L1` if the value in register `$r1` equals the value in register `$r2`
  - **beq** is "**b**ranch **if** **e**qual"
  - C code: `if (a == b) goto L1`
- **bne $r1, $r2, L1**
  - Go to statement labeled `L1` if the value in register `$r1` does not equal the value in register `$r2`
  - **bne** is "**b**ranch **if** **n**ot **e**qual"
  - C code: `if (a != b) goto L1`

## 3.2 Unconditional Jump **j**

- Processor **always** follows the branch
- **j L1**
  - Jump to label `L1` unconditionally
  - C code: `goto L1`
- Technically equivalent to such statement

```
beq $s0, $s0, L1
```

```
    beq $s3, $s4, L1
    j   Exit
L1: add $s0, $s1, $s2
Exit:
```

NOTE:
The general technique of compiling from C to MIPS with selection/repetition is to first remove the construct and replace it with either one of these:

```
if (x == y) goto L;  // this is beq
if (x != y) goto L;  // this is bne
```

Some tricks can be used (e.g., inversion) to generate fewer lines of codes.

IF statements

## 3.3 IF statement (1/2)

| C Statement to translate | Variables Mapping |
|---|---|
| `if (i == j)`<br>`  f = g + h;` | f → $s0<br>g → $s1<br>h → $s2<br>i → $s3<br>j → $s4 |

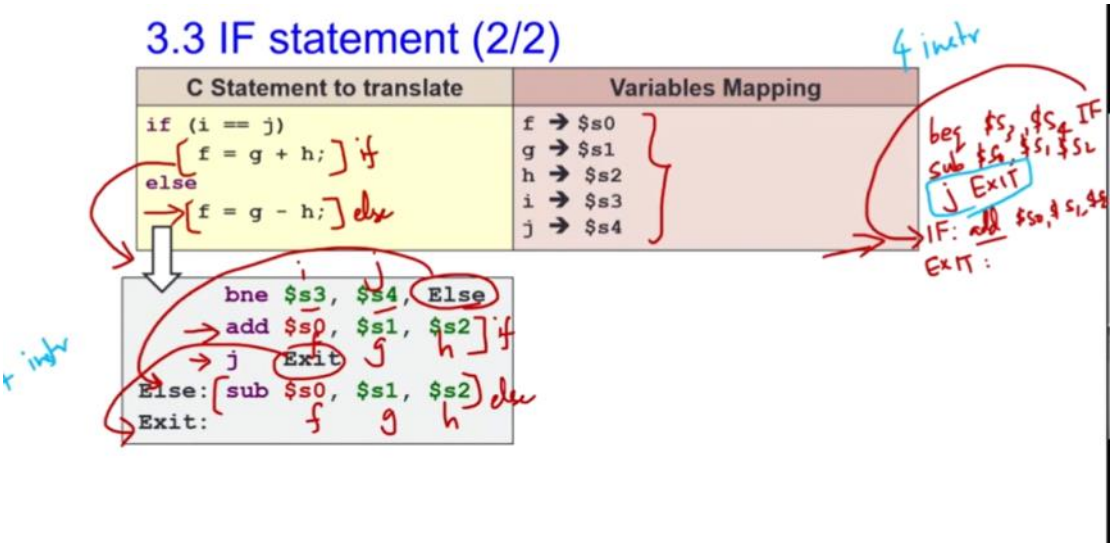
```
    beq $s3, $s4, L1
    j   Exit
L1: add $s0, $s1, $s2
Exit:
```

3 instr.

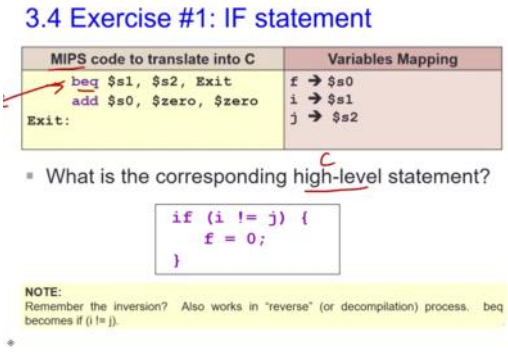
```
    bne $s3, $s4, Exit
    add $s0, $s1, $s2
Exit:
```

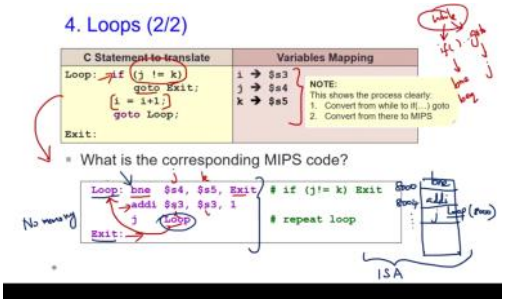
2 instr

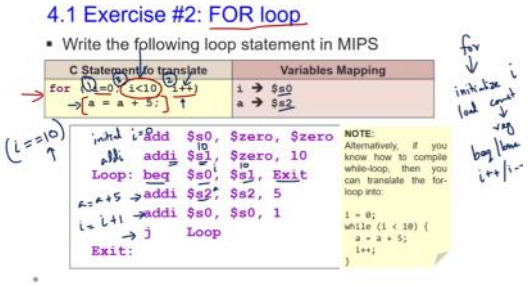
Bne is more efficient here




Both casess are similar - cuz using 4 instruction



## 4.2 Inequalities (1/2)

- We have **beq** and **bne**, what about branch-if-less-than?
  - There is no real **blt** instruction in MIPS
- Use **slt** (set on less than) or **slti**.

```
slt $t0, $s1, $s2
```

=

```
if ($s1 < $s2)
    $t0 = 1;
else
    $t0 = 0;
```

**NOTE:**
If we use the "ternary operator" discussed in tutorial, then:

```
$t0 = ($s1 < $s2) ? 1 : 0;
```


## 4.2 Inequalities (1/2)

- We have **beq** and **bne**, what about branch-if-less-than?
  - There is no real **blt** instruction in MIPS
- Use **slt** (set on less than) or **slti**.

```
slt $t0, $s1, $s2
```

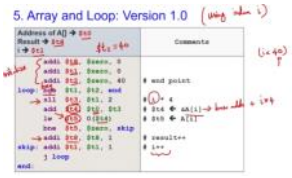
=

```
if ($s1 < $s2)
    $t0 = 1;
else
    $t0 = 0;
```

**NOTE:**
If we use the "ternary operator" discussed in tutorial, then:

```
$t0 = ($s1 < $s2) ? 1 : 0;
```

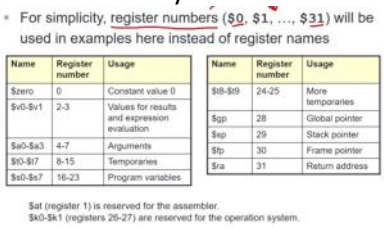

Array and Loop

- To access arr elements in a loop
  - Initialization for result variable, loop counter, and array pointer
  - Label:
    - Work by
      - Calculating address
      - Load data
      - Perform task
    - Update loop counter and arr pointers
    - Compare and branch
- Count number of zeros in 40 elem word arr
- 

- 

# Week 4

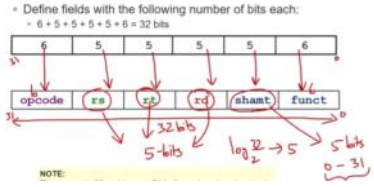
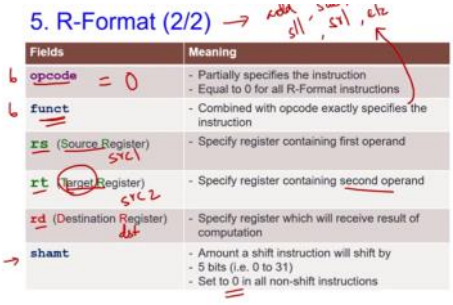
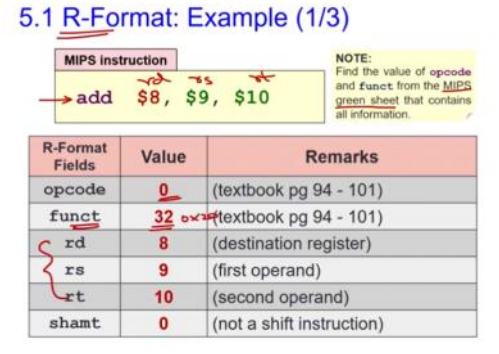
Tuesday, September 5, 2023 9:24 PM

- Each mips instruction has fixed length of 32 bits
- MIPS instruction classification: R, I, J
  - Classified according to operands
  - **R-format** register format: op $r1 $r2 $r3
    - Instructions which use 2 source registers and 1 destination register
      - Add, sub, and, or , nor, slt, etc
      - Srl, sll
  - **I format** - intermediate format: op $r1, 4r2, Immd
    - Instructions which use 1 source register, 1 immediate value ad 1 destination register
      - Addi, andi, ori, slti, lw, sw, beq, bne
  - **J-format** (jump format: op Immd
    - Uses only 1 immediate val
  - 
    

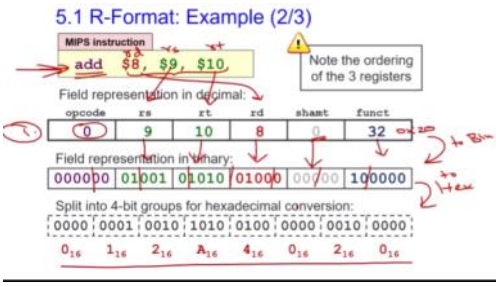
| Name | Register number | Usage |
|---|---|---|
| $zero | 0 | Constant value 0 |
| $v0-$v1 | 2-3 | Values for results and expression evaluation |
| $a0-$a3 | 4-7 | Arguments |
| $t0-$t7 | 8-15 | Temporaries |
| $s0-$s7 | 16-23 | Program variables |

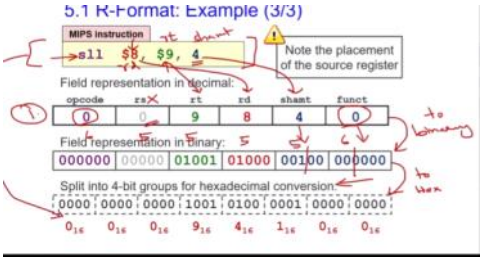
| Name | Register number | Usage |
|---|---|---|
| $t8-$t9 | 24-25 | More temporaries |
| $gp | 28 | Global pointer |
| $sp | 29 | Stack pointer |
| $fp | 30 | Frame pointer |
| $ra | 31 | Return address |

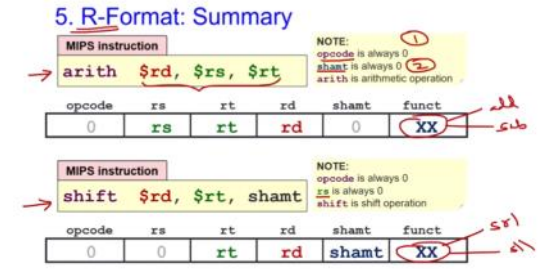
R- FORMAT

- 
  
- 
  
- R- format opcode is always 0
- 
  

| R-Format Fields | Value | Remarks |
|---|---|---|
| opcode | 0 | (textbook pg 94 - 101) |
| funct | 32 | (textbook pg 94 - 101) |
| rd | 8 | (destination register) |
| rs | 9 | (first operand) |
| rt | 10 | (second operand) |
| shamt | 0 | (not a shift instruction) |

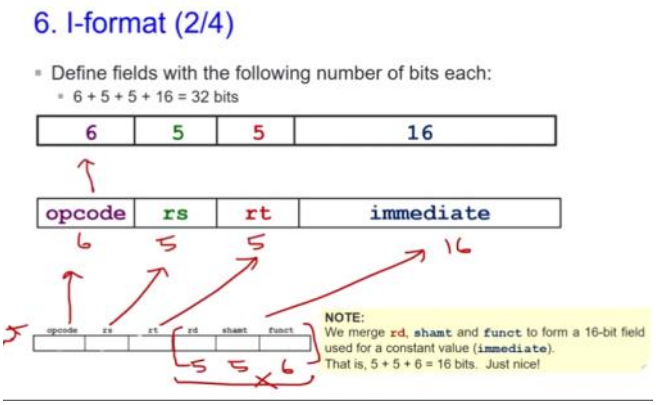
- 

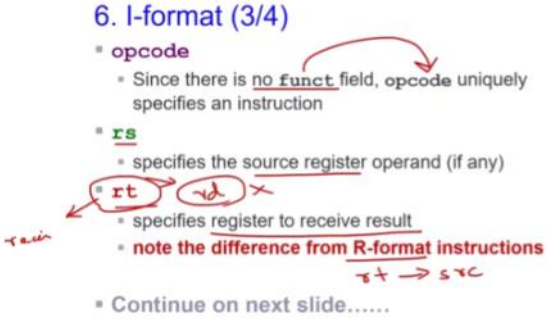



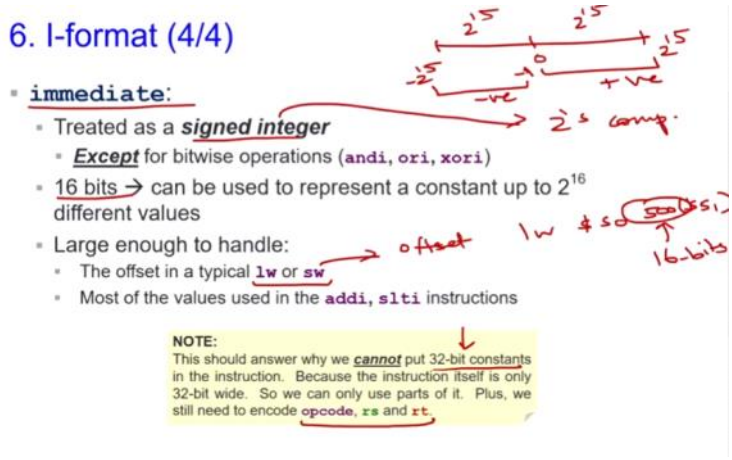

- 

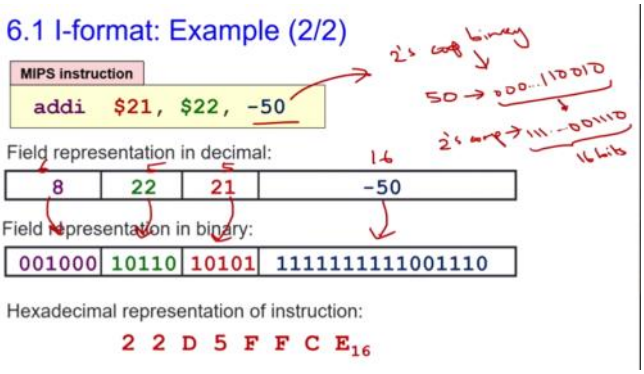

I Format Instruction

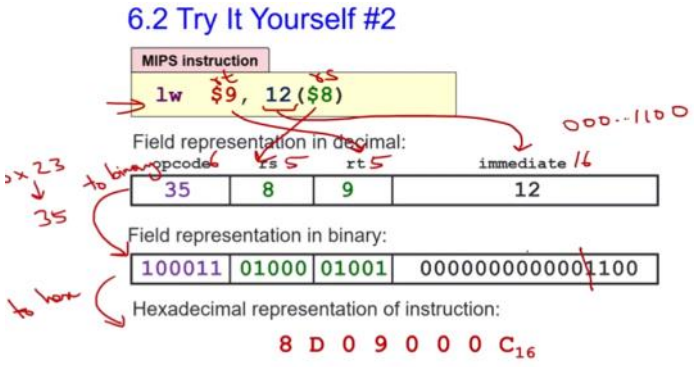
- Instructions with immediate vals
- 5 bit shamt field can only represent 0 to 31
- Immediates can be much larger

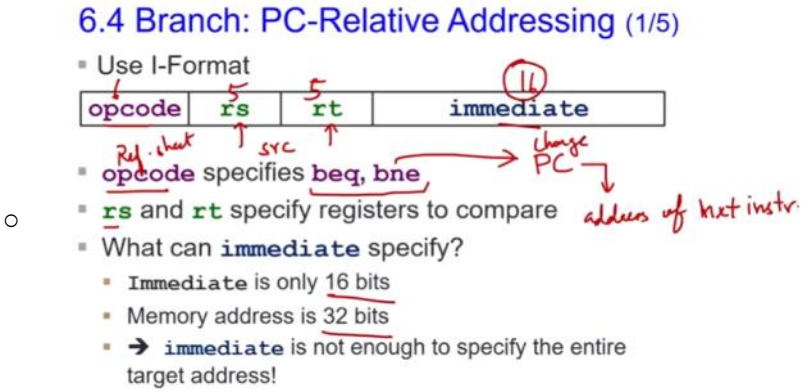
- 


- 

- 

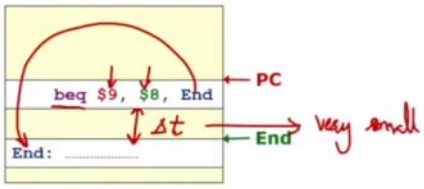
- 

- Need to add preceding zeros
- 

- Instructions are 32 bit long and stored in memory
- Word aligned
- PROGRAM COUNTER (PC)
  - A special register that keeps address of next instruction being executed in processor
  - 

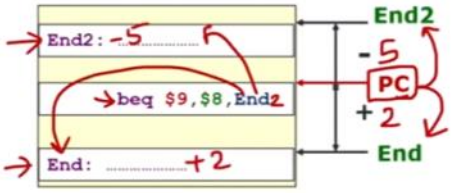
## 6.4 Branch: PC-Relative Addressing (2/5)

- How do we usually use branches?
  - **Answer:** `if-else`, `while`, `for` (bne, beq)
  - Loops are generally **small**:
    - Typically up to 50 instructions
  - Unconditional jumps are done using jump instructions (`j`), not the branches
  - **Conclusion:** A branch often changes **PC** by a small amount




## 6.4 Branch: PC-Relative Addressing (3/5)

- **Solution:**
  - Specify target address **relative to the PC**
  - Target address is generated as: PC = PC + immed
    - PC + the 16-bit **`immediate`** field
    - The **`immediate`** field is a signed two's complement integer
- ➔ Can branch to ± $2^{15}$ bytes from the PC:
  - Should be enough to cover most loops




## 6.4 Branch: PC-Relative Addressing (5/5)

- Branch calculation:

If the branch is **not taken**:
**PC = PC + 4**
(**PC + 4** is address of next instruction)

If the branch is **taken**:
**PC = (PC + 4) + (`immediate` × 4)**

  - Observations:
    - **`immediate`** field specifies the number of words to jump, which is the same as the number of instructions to "skip over"
    - **`immediate`** field can be positive or negative
    - Due to hardware design, add **`immediate`** to (PC+4), not to PC (more in later topic)

## 6.5 Branch: Example (1/3)

```
Loop:  beq  $9, $0, End   # rlt addr: 0
       add  $8, $8, $10   # rlt addr: 4
       addi $9, $9, -1    # rlt addr: 8
       j    Loop          # rlt addr: 12
End:                      # rlt addr: 16
```

- beq is an I-Format instruction →

| I-Format Fields | Value | Remarks |
|---|---|---|
| opcode | 4 | |
| rs | 9 | (first operand) |
| rt | 0 | (second operand) |
| immediate | ??? | (in base 10) |

# instructions ?

- Immediate is # of instruction to skip
- # of instuctions get counted from the instruction following the branch

## 6.5 Branch: Example (3/3)

```
Loop:  beq  $9, $0, End   # rlt addr: 0
       add  $8, $8, $10   # rlt addr: 4
       addi $9, $9, -1    # rlt addr: 8
       j    Loop          # rlt addr: 12
End:                      # rlt addr: 16
```

- Field representation in decimal:

| opcode | rs | rt | immediate |
|---|---|---|---|
| 4 | 9 | 0 | 3 |

To bin

Field representation in binary:

| 000100 | 01001 | 00000 | 0000000000000011 |
|---|---|---|---|

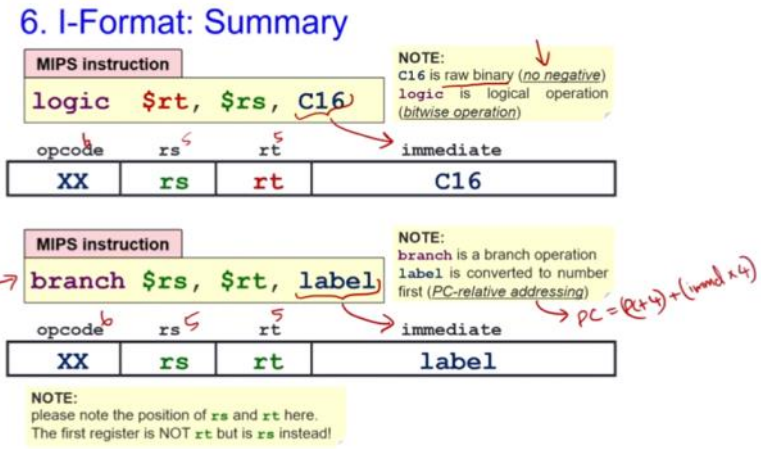
to hex

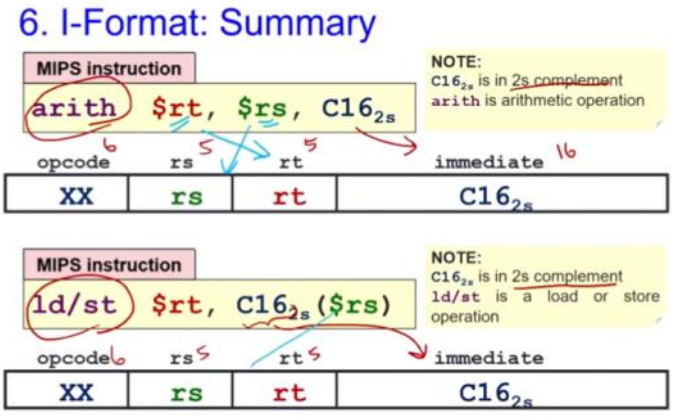
## 6.6 Try It Yourself #3

```
Loop:  beq  $9, $0, End   # rlt addr: 0
       add  $8, $8, $10   # rlt addr: 4
       addi $9, $9, -1    # rlt addr: 8
       beq  $0, $0, Loop  # rlt addr: 12
End:                      # rlt addr: 16
```

- What would be the immediate value for the second beq instruction? ?

-4



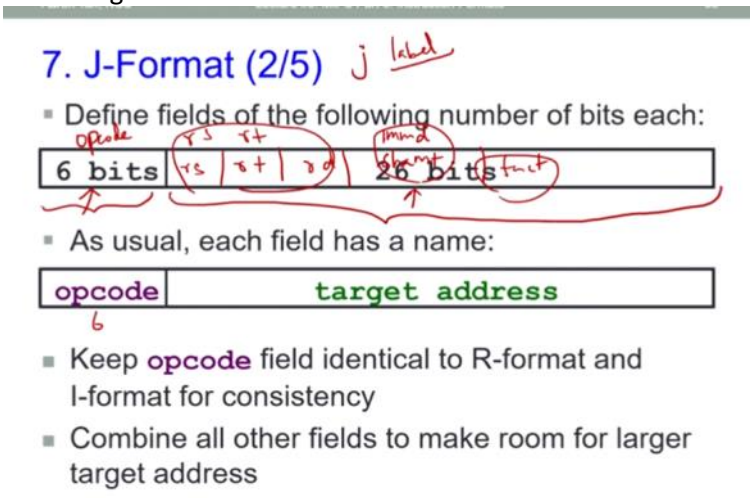







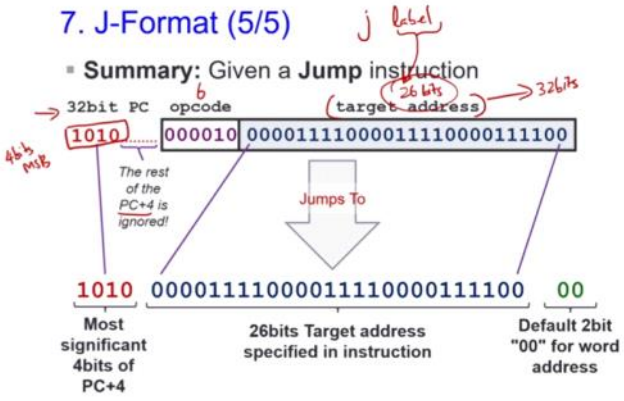
---

- For branches we used PC relative addressing but the max we can jump is 32768 (2^15). This is ok for branches since we don't need to branch too far
- But for general jumps, we can go anywhere in momory
- We however, cannot write whole address. Because we need to use 6 bits for opcode. Then the remaining 26 bits

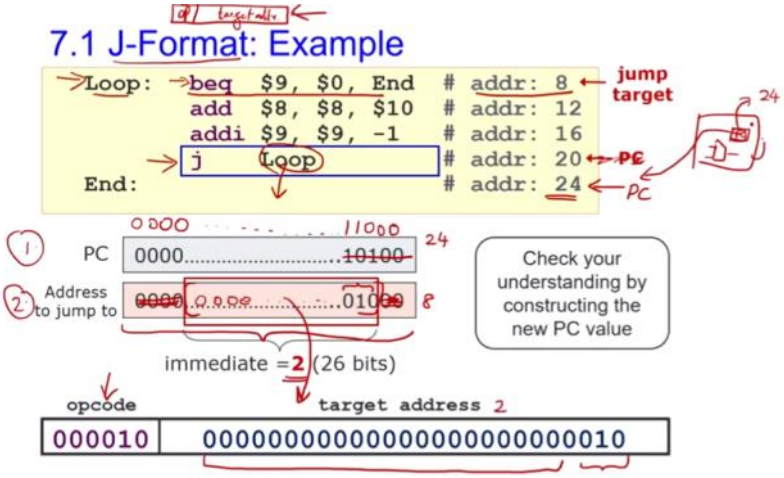



- Since we can only specify 26 bits of a 32 bit address
- Like branches, jumps only jump to word-aligned addresses so last 2 bits are always 00 (multiples of 4)
- So we assume address ends with 00 and leave it out

- So we can specify 28 bits of 32 bit address
- Then for remaining 4 bits, we use the 4 most significant bits of next address
- Can jump 2^26 instructions
- Maximum jump range is 256 mb boundary (2^28 bytes)
- If program straddles 256mb boundary, there is jr instruction. But out of syllabus




- Effective address




- To get the actual/effective address (32 bits), take target address(26 bits) add 2 zeros to the end and add the MSB from PC to start

## 7.2 Branching Far Way

- Given the instruction

  `beq $s0, $s1, L1`

  Assume that the address `L1` is farther away from the PC than what can be supported by `beq` and `bne` instructions

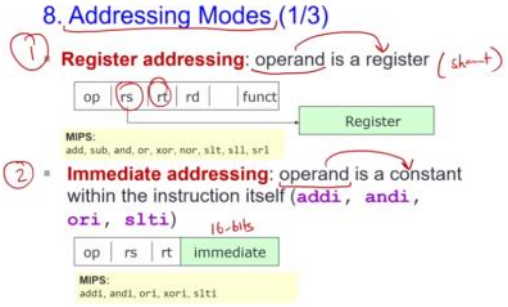
- **Challenge:**
  - Construct an equivalent code sequence with the help of unconditional (`j`) and conditional branch (`beq, bne`) instructions to accomplish this far away branching

NOTE: Discuss in forum

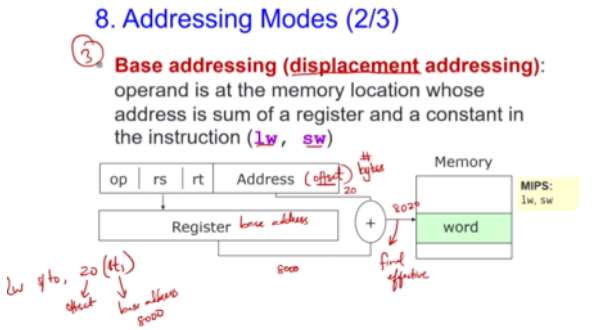
```
Bne $s0, $s1, End
        j L1
End
```

ADDRESSING MODES

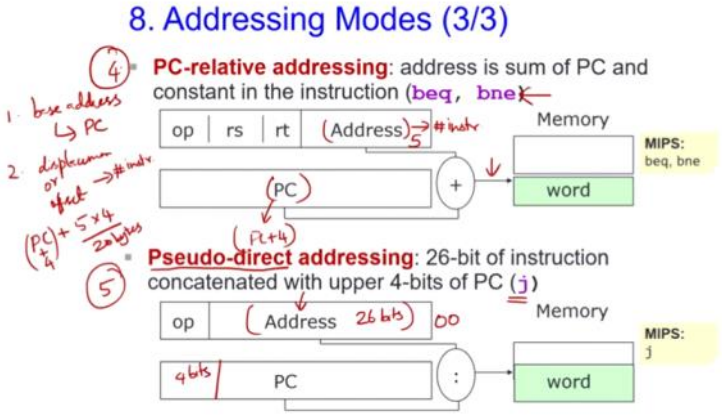
- Register addressing
  - Operand is register
- Immediate addressing
  - Operand is constant within the instruction itself




- Base addressing (displacement addressing)
  - Operand is at the memory location whose address is sum of a register and a constant in the instruction (lw, sw)
  - Displacement is # of bytes




- PC-relative addressing
  - Address is sum of PC and constant in instruction
  - Similar to Base addressing HOWEVER
    - Base address is now stored in PC
    - Displacement//offset is not #of bytes but # of instructions
  - Rmb PC is actually PC+4
- Pseudo-direct addressing
  - 26 bit of instruction concatenated with upper 4 bits of pc (j)



## Summary (1/2)

- MIPS Instruction:
  32 bits representing a single instruction

| | | | | | | |
|---|---|---|---|---|---|---|
| R | opcode | rs | rt | rd | shamt | funct |
| I | opcode | rs | rt | immediate | | |
| J | opcode | target address | | | | |

- Branches and load/store are both I-format instructions; but branches use PC-relative addressing, whereas load/store use base addressing
- Branches use ***PC-relative*** addressing
  Jumps use ***pseudo-direct*** addressing
- Shifts use R-format, but other immediate instructions (`addi`, `andi`, `ori`) use I-format

**IN OTHER WORDS:**
**beq, bne**: ignore PC, count displacement
**j**: use PC, ignore displacement

## Summary (2/2)

**MIPS assembly language**

| Category | Instruction | Example | Meaning | Comments |
|---|---|---|---|---|
| Arithmetic | add | `add $s1, $s2, $s3` | `$s1 = $s2 + $s3` | Three operands; data in registers |
| | subtract | `sub $s1, $s2, $s3` | `$s1 = $s2 - $s3` | Three operands; data in registers |
| | add immediate | `addi $s1, $s2, 100` | `$s1 = $s2 + 100` | Used to add constants |
| Data transfer | load word | `lw $s1, 100($s2)` | `$s1 = Memory[$s2 + 100]` | Word from memory to register |
| | store word | `sw $s1, 100($s2)` | `Memory[$s2 + 100] = $s1` | Word from register to memory |
| | load byte | `lb $s1, 100($s2)` | `$s1 = Memory[$s2 + 100]` | Byte from memory to register |
| | store byte | `sb $s1, 100($s2)` | `Memory[$s2 + 100] = $s1` | Byte from register to memory |
| | load upper immediate | `lui $s1, 100` | $s1 = 100 * $2^{16}$ | Loads constant in upper 16 bits |
| Conditional branch | branch on equal | `beq $s1, $s2, 25` | if (`$s1 == $s2`) go to PC + 4 + 100 | Equal test; PC-relative branch |
| | branch on not equal | `bne $s1, $s2, 25` | if (`$s1 != $s2`) go to PC + 4 + 100 | Not equal test; PC-relative |
| | set on less than | `slt $s1, $s2, $s3` | if (`$s2 < $s3`) `$s1` = 1; else `$s1` = 0 | Compare less than; for beq, bne |
| | set less than immediate | `slti $s1, $s2, 100` | if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0 | Compare less than constant |
| Uncondi-tional jump | jump | `j 2500` | go to 10000 | Jump to target address |
| | jump register | `jr $ra` | go to `$ra` | For switch, procedure return |
| | jump and link | `jal 2500` | `$ra` = PC + 4; go to 10000 | For procedure call |

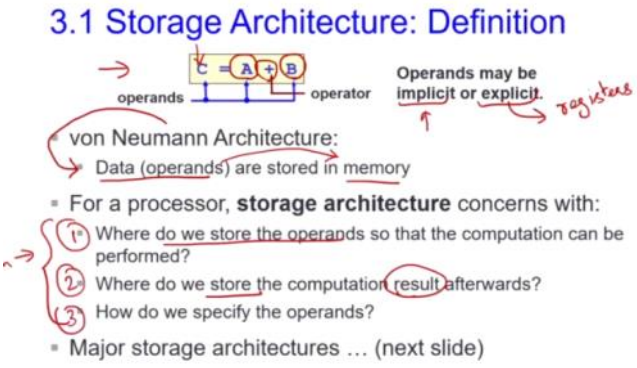
ISA

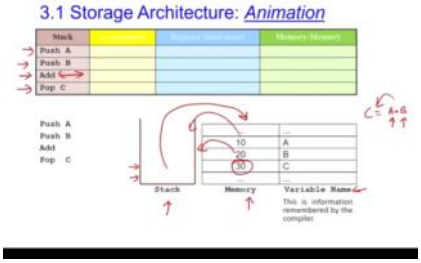
- There are 2 major design philosophy for ISA
- CISC and RISC
- Complex Instruction Set Computer CISC
  - Intel x86-32
  - Single instruction performs complex operation
    - VAX archi had an instruction to multiply polynomials
    - VAX is  a mini computer used yrs ago
  - There was smaller program size as memory was premium
  - So they try to optimize memory

    - Complex implementation, no room for hardware optimization
- Reduced Instruction Set Computer (RISC)
    - MIPS, ARM
    - Keeps the instruction set small and simple, makes it easier to build/optimize hardware
    - Burden on software to combine simpler operations to implement high-level language statements
    - More and cheaper memory now
    - And it allows compiler optimization
    - These days RISC is used more

- Concepts in ISA design
    - Data Storage - where we get data
    - Memory Addressing Modes - how to access data from memory
    - Operations in instruction set - types operations
    - Instruction format - types of instruction formats
    - Encoding the instruction set - how instructions are encoded into binary bits

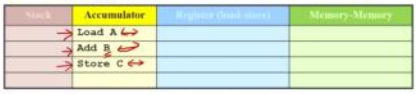
1. DATA Storage

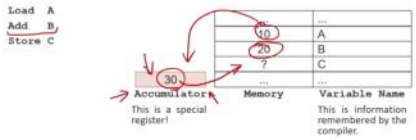
- In MIPS, general purpose register architecture
- 


- **Common designs**
- Stack Architecture
    - Operands are implicitly on top of the stack
    - LIFO
    - Has push/pop operations
    - 


- Accumulator architecture
    - One operand is implicitly in the accumulator (a special register)
    - IBM 701 example

3.1 Storage Architecture: *Animation*

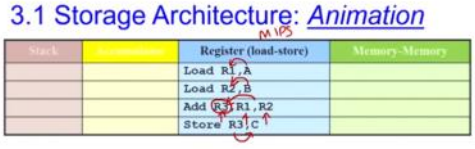


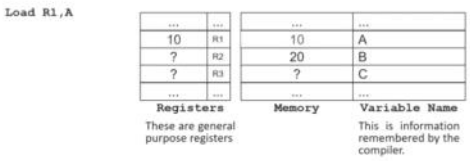




---

- General-purpose register architecture
  - Only explicit operands
  - Register memory architecture
    - Where 1 operand in memory
      - Ex. Intel 80386
  - Register-register (or load store) architecture
    - MIPS/ARM

3.1 Storage Architecture: *Animation*




- Memory-memory architecture
  - All operands in memory
  - Like the mini computers used in 70s

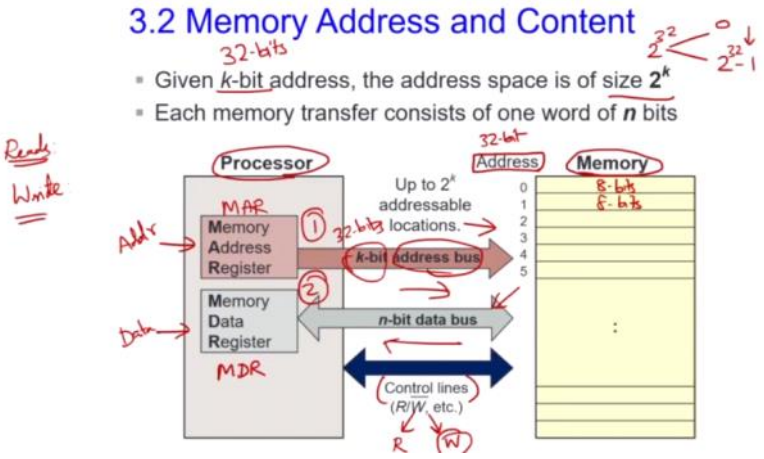
3.1 Storage Architecture: Example

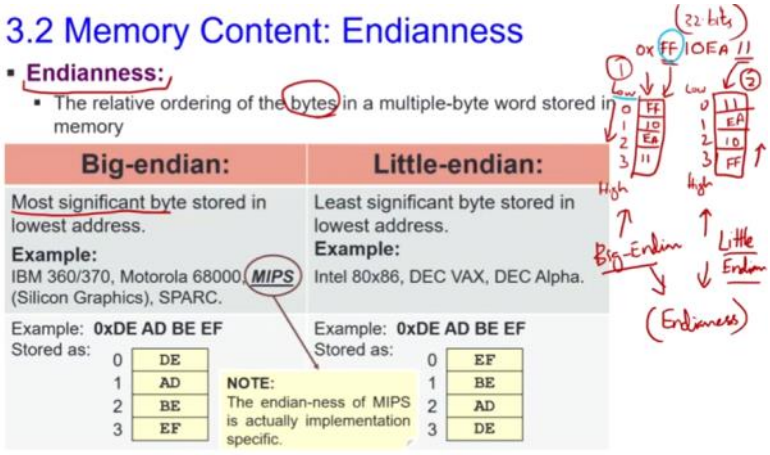
| Stack | Accumulator | Register (load-store) | Memory-Memory |
|---|---|---|---|
| Push A | Load A | Load R1,A | Add C, A, B |
| Push B | Add B | Load R2,B | |
| Add | Store C | Add R3,R1,R2 | |
| Pop C | | Store R3,C | |


- Modern processors use general purpose register (GPR_
- RISC computers use register-register (load/store) design
  - Coz simplify hardware design
  - Balanced pipeline
- CISC computers use a mixture of register-register and register-memory

2. Memory Addressing Mode

- Given k bit address, address space is of size 2^k
- Each memory transfer consists of 1 word of n bits
- Memory address register - stores address which the processor needs to read
- Memory Data register - stores data that needs to be written/read into memory
- Address bus is always unidirectional

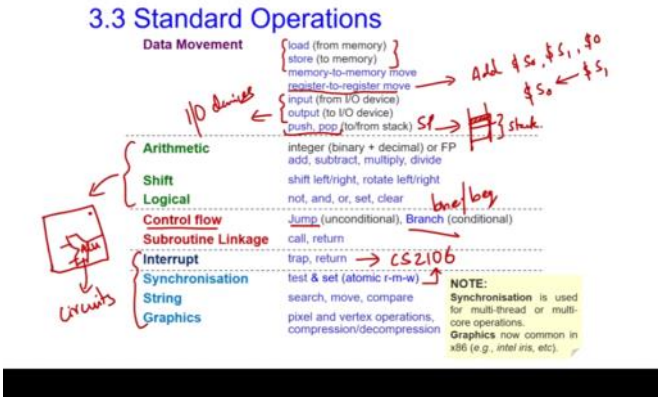
- Data bus can go both ways
- There are also control lines - manages whether current memory access is Read/Write
  - 

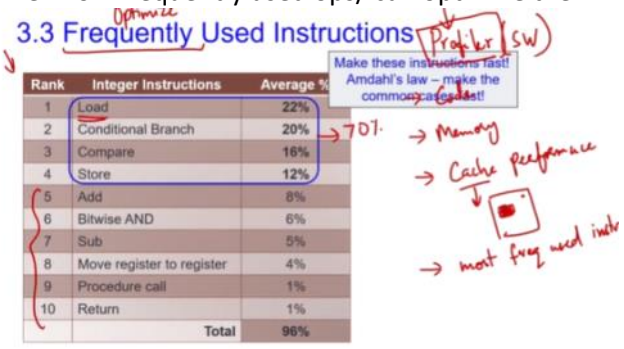
- ENDIANNESS - relative ordering of bytes in multilple-byte word stored in memory
  - Big endian - LSB -> MSB
  - Little endian - MSB -> LSB
  - 


- Addressing modes - ways to specify an operand in assembly
  - In mips, there are only 3
    - Register -> operand is in a register (add $t1, $t2, $t3
    - Immediate -> addi
    - Displacement -> lw
  - 

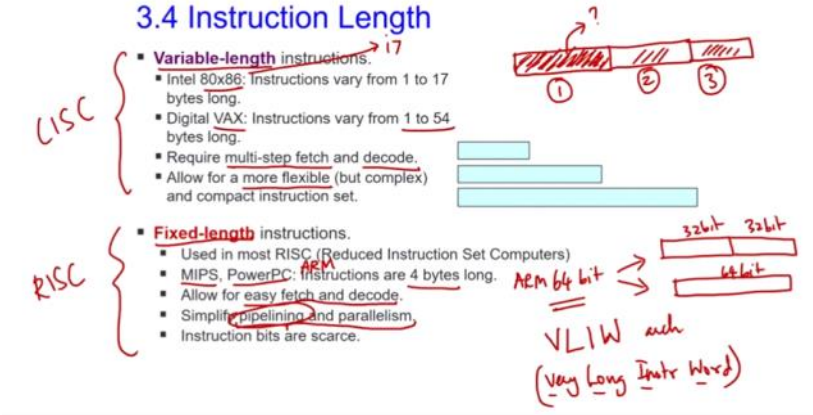

OPERAtions in the instruction set

- Standard operations in an instruction set

- Frequently used instruction
- 

- Besides string/graphics, RISC processor supports all of above
- If we know frequently used ops/ can optimize them
- 


Instruction Formats

- Instruction length
  - In mips, they are 32 bits
  - For 64 bit machines, instructions are 64 bit
  - 

- Instruction Fields
  - Type and size of operands
  - Consists of opcode and operands
    - Opcode: unique code to specify desired op
    - Operands: zero or more additional information needed for op

## 3.4 Instruction Fields

- An instruction consists of
  - **opcode**: unique code to specify the desired operation
  - **operands:** zero or more additional information needed for the operation
- The operation designates the type and size of the operands
  - **Typical type and size:** Character (8 bits), half-word (eg: 16 bits), word (eg: 32 bits), single-precision floating point (eg: 1 word), double-precision floating point (eg: 2 words).
- Expectations from any new 32-bit architecture:
  - Support for 8-, 16- and 32-bit integer and 32-bit and 64-bit floating point operations. A 64-bit architecture would need to support 64-bit integers as well.

Encoding the instruction set

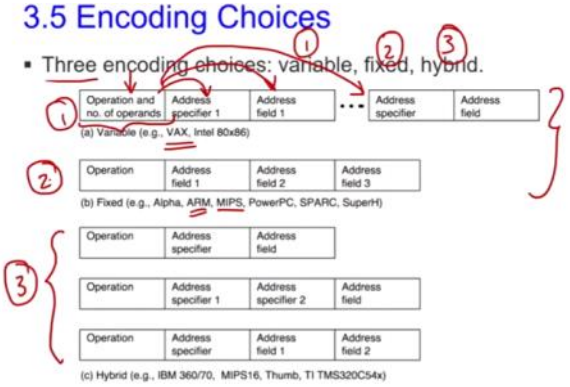
- MIPS has 32 registers
  - Need 5 bits for register indexing

## 3.5 Instruction Encoding: Overview

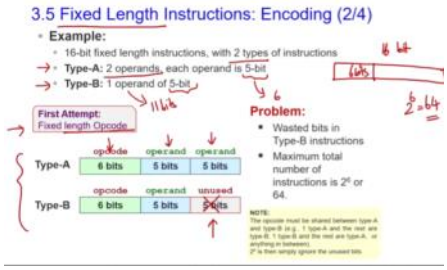
- How are instructions represented in binary format for execution by the processor?
- Issues:
  - Code size, speed/performance, design complexity.
- Things to be decided:
  - Number of registers
  - Number of addressing modes
  - Number of operands in an instruction
- The different competing forces:
  - Have many registers and addressing modes
  - Reduce code size
  - Have instruction length that is easy to handle (fixed-length instructions are easier to handle)

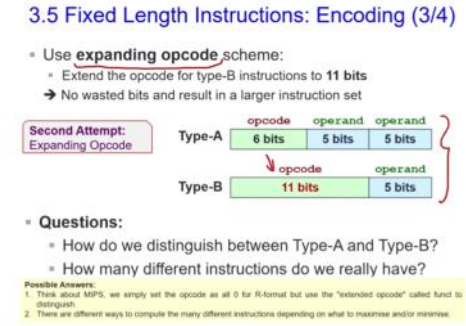
## 3.5 Encoding Choices

- Three encoding choices: variable, fixed, hybrid.




- Fixed Length : encoding
  - How can we fit multiple sets of instruction types into same number of bits
  - Expanding Opcode scheme
    - Opcode has variable lengths for diff instructions
    - To maximize instruction bits

- 

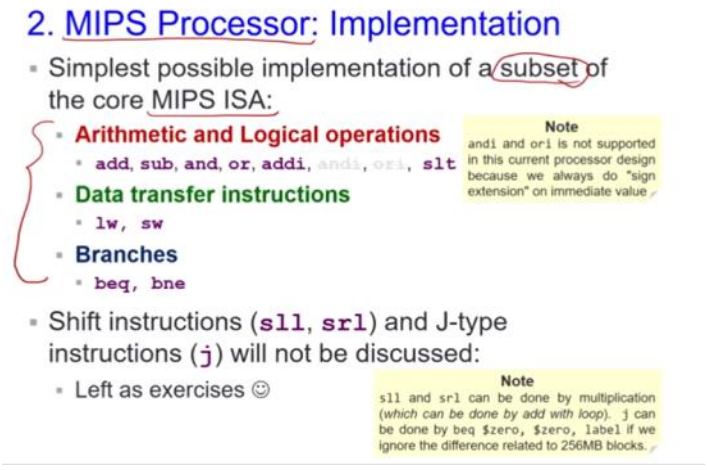




  - We reserve 1 6 bit number to be assigned to type b
  - 2^6-1 type a opcodes possible
  - 111111
  - The 11 bits of type b will always start with 111111
  - So we have only 2^5 opcodes of type b
  - 11111100000 -> 11111111111

- We maximize # of instruction for type b and minimize for type A
- So opcode for type a will be only 1: 000000
- Type B: (2^6-1) * 2^5 instructions
- Type A: only 1 instruction

# Week 5

Monday, September 18, 2023 11:41 PM

- C -> Assembly -> embedded binary -> processor
- Building a processor has 2 parts Datapath and control path
  - Data path is collection of components that process data
  - Performs arithmetic , logical, memory ops
  - Control tells the Datapath, memory and I/O devices what to do according to program instructions
  - 


- Executing MIPS instruction overview
  - Program is stored in memory
  - The processor has a Program Counter (PC)
    - Which is address of next instruction to be executed
  - Each instruction is executed in 5 stages: fetch, decode, operand fetch, execute, result write (store)
  - Fetch
    - Getting instruction from memory
    - Address is in PC register
  - Decode
    - Find out the operation required
  - Operand fetch
    - Get operand needed for operation
  - Execute
    - Perform operation
  - Result write
    - Store result

## 4. MIPS Instruction Execution (1/2)

- Show the actual steps for 3 representative MIPS instructions
- Fetch and Decode stages not shown:
  - The standard steps are performed

| | add $rd, $rs, $rt | lw $rt, ofst($rs) | beq $rs, $rt, ofst |
|---|---|---|---|
| Fetch / Decode | *standard* | *standard* | *standard* |
| Operand Fetch | ○ Read [**$rs**] as *opr1*<br>○ Read [**$rt**] as *opr2* | ○ Read [**$rs**] as *opr1*<br>○ Use **ofst** as *opr2* | ○ Read [**$rs**] as *opr1*<br>○ Read [**$rt**] as *opr2* |
| Execute | *Result* = *opr1* + *opr2* | ○ *MemAddr* = *opr1* + *opr2*<br>○ Use *MemAddr* to read from memory | *Taken* = (*opr1* == *opr2*)?<br>*Target* = (**PC**+4) + **ofst**×4 |
| Result Write | *Result* stored in **$rd** | *Memory* data stored in **$rt** | if (*Taken*)<br>**PC** = *Target* |

- **opr** = operand
- **MemAddr** = address
- **ofst** = offset

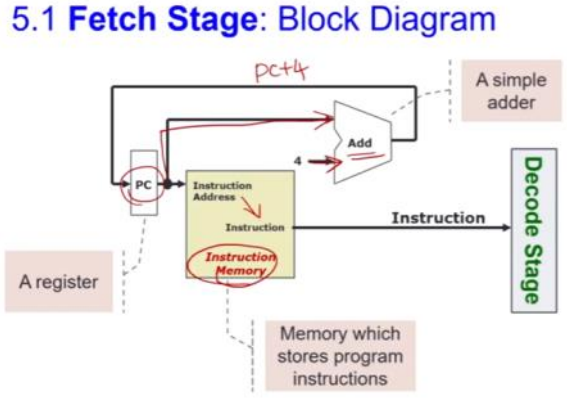
## 4. MIPS Instruction Execution (2/2)

- Design changes:
  - Merge *Decode* and *Operand Fetch* – Decode is simple for MIPS
  - Split *Execute* into ***ALU*** (Calculation) and ***Memory Access***

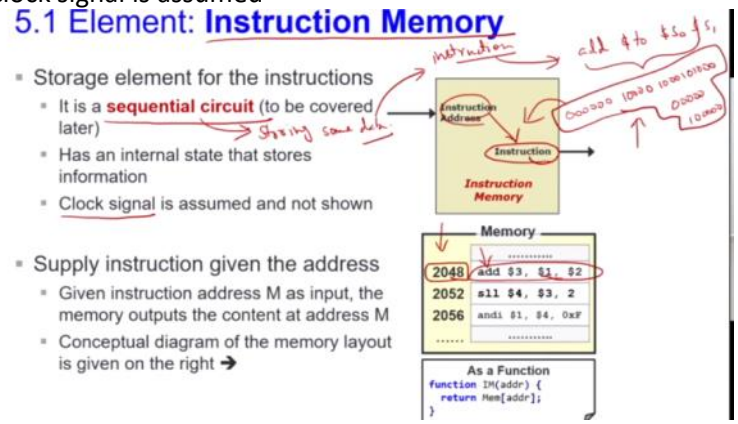
| | add $rd, $rs, $rt | lw $rt, ofst($rs) | beq $rs, $rt, ofst |
|---|---|---|---|
| Fetch | Read instr from [PC] | Read instr from [PC] | Read instr from [PC] |
| Decode / Operand Fetch | ○ Read [**$rs**] as *opr1*<br>○ Read [**$rt**] as *opr2* | ○ Read [**$rs**] as *opr1*<br>○ Use **ofst** as *opr2* | ○ Read [**$rs**] as *opr1*<br>○ Read [**$rt**] as *opr2* |
| ALU | *Result* = *opr1* + *opr2* | *MemAddr* = *opr1* + *opr2* | *Taken* = (*opr1* == *opr2*)?<br>*Target* = (**PC**+4) + **ofst**×4 |
| Memory Access | | Use *MemAddr* to read from memory | |
| Result Write | *Result* stored in **$rd** | *Memory* data stored in **$rt** | if (*Taken*)<br>**PC** = *Target* |

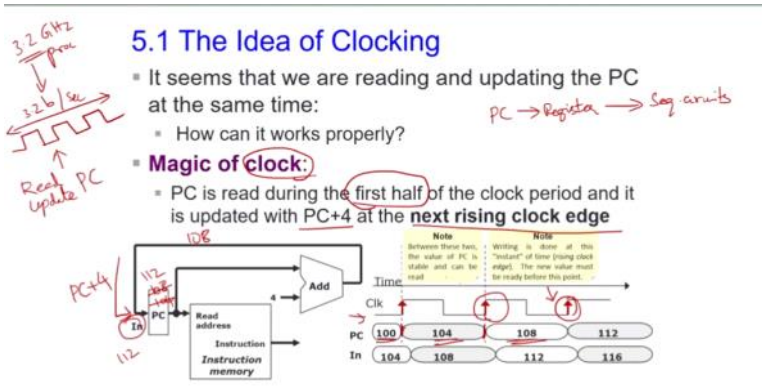
Fetch Stage

- Requirements
  - Use PC to fetch instruction from memory
    - PC is special register in processor
  - Increment PC by 4 to get address of next instruction
    - 32 bits of each MIPS instruction = 4 bytes
    - Expect branch/jump is different
  - Output to the decode stage



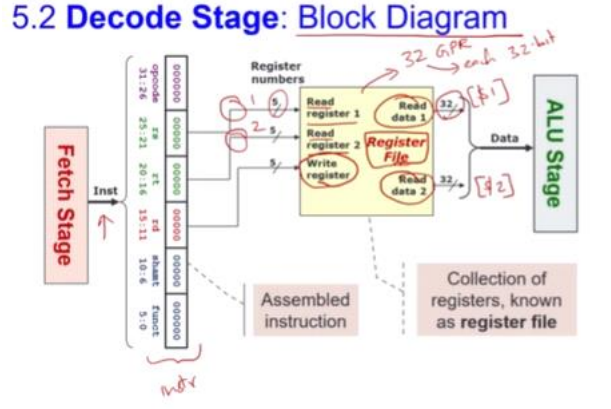

  - INSTRUCTIONAL MEMORY
    - Storage element for the instructions
    - A sequential circuit
    - Has an internal state that stores information

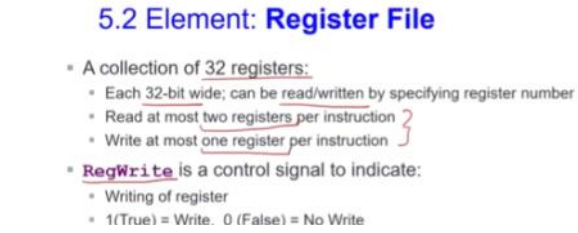
- Clock signal is assumed
- 

- 


Decode stage

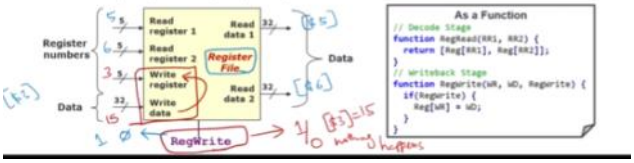
- Gather data from instruction fields
- Read opcode to determine instruction type and field lengths
- Read data from all necessary registers
  - Ex. (add -> 2, addi -> 1, j -> 1
- Input from previous stage fetch
- The register file, is basically if you input the register #, it will output the register value

- 

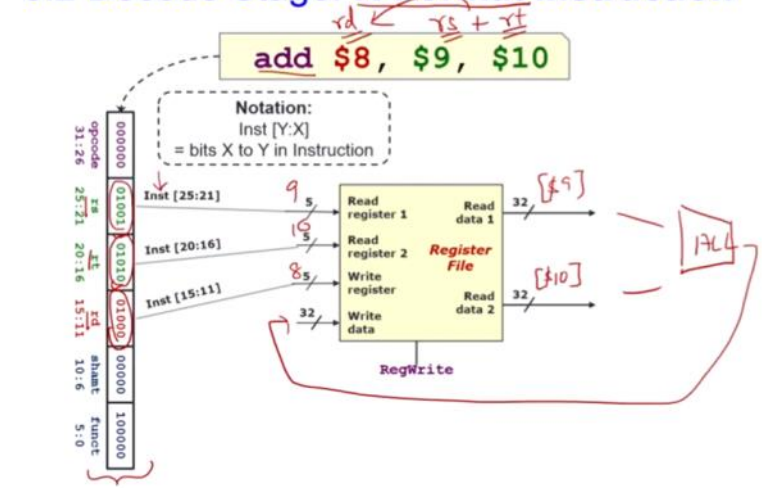
- 

## 5.2 Element: **Register File**

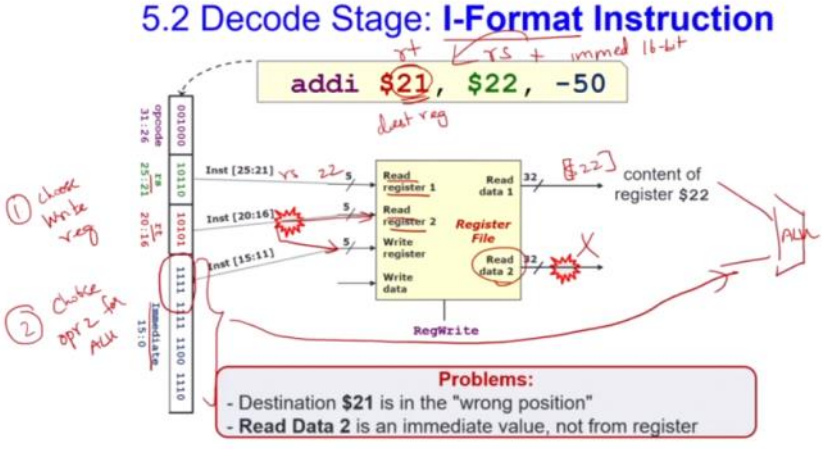
- A collection of 32 registers:
  - Each 32-bit wide; can be read/written by specifying register number
  - Read at most two registers per instruction
  - Write at most one register per instruction
- `RegWrite` is a control signal to indicate:
  - Writing of register
  - 1(True) = Write, 0 (False) = No Write

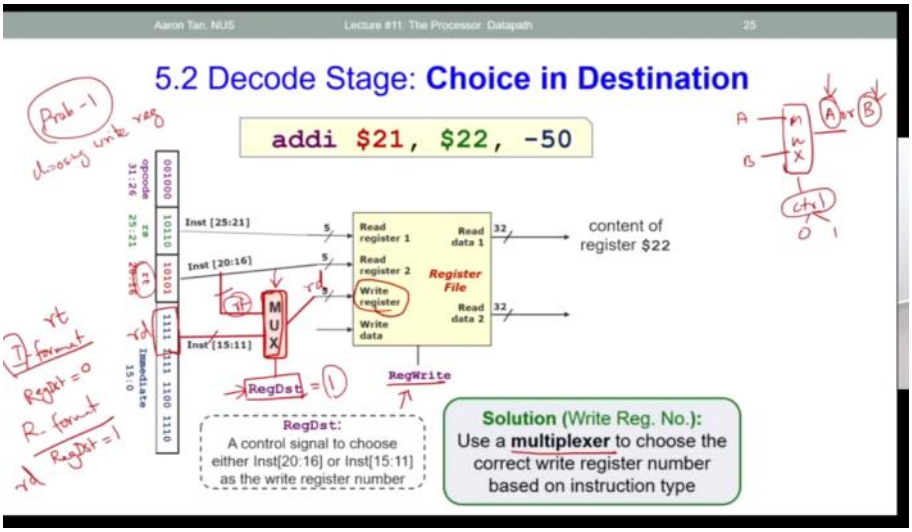



## 5.2 Decode Stage: **R-Format Instruction**

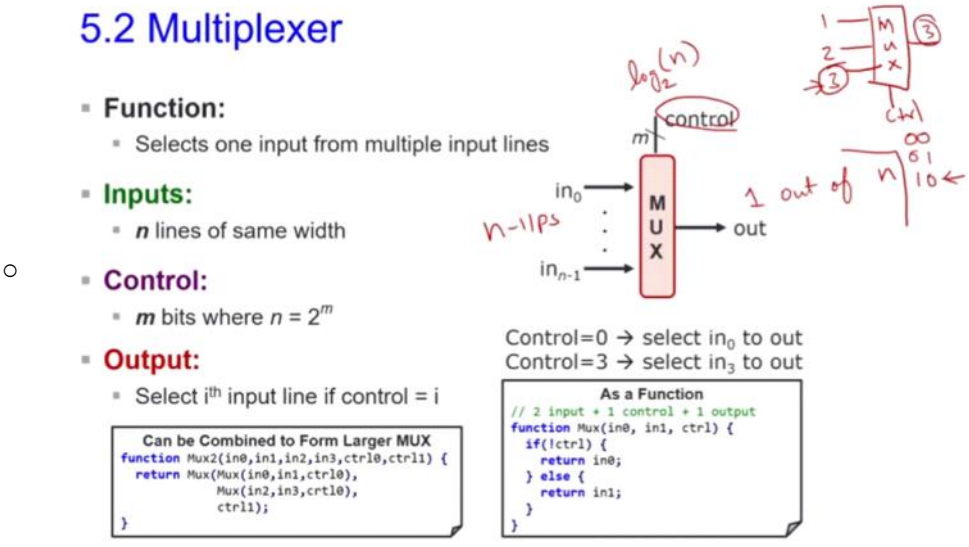


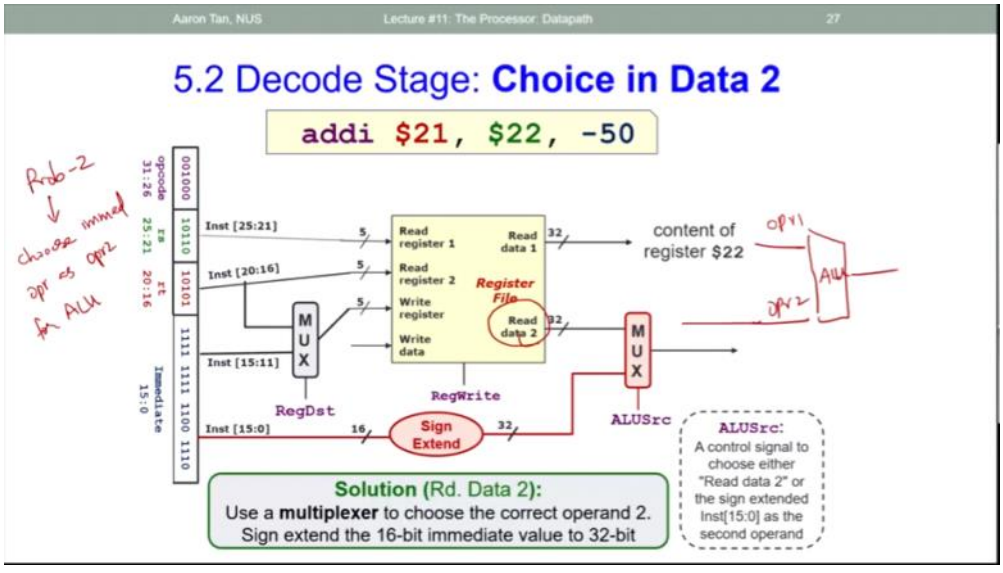

## 5.2 Decode Stage: **I-Format Instruction**





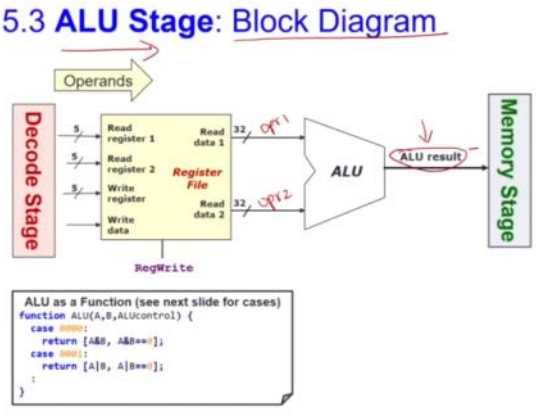

- Multiplexer
  - Functions to select 1 input from multiple input lines
  - To choose correct write register based on instruction type
  - For I-Format instructions, multiplexer slhd choose 0
    - For R-format, choose 1

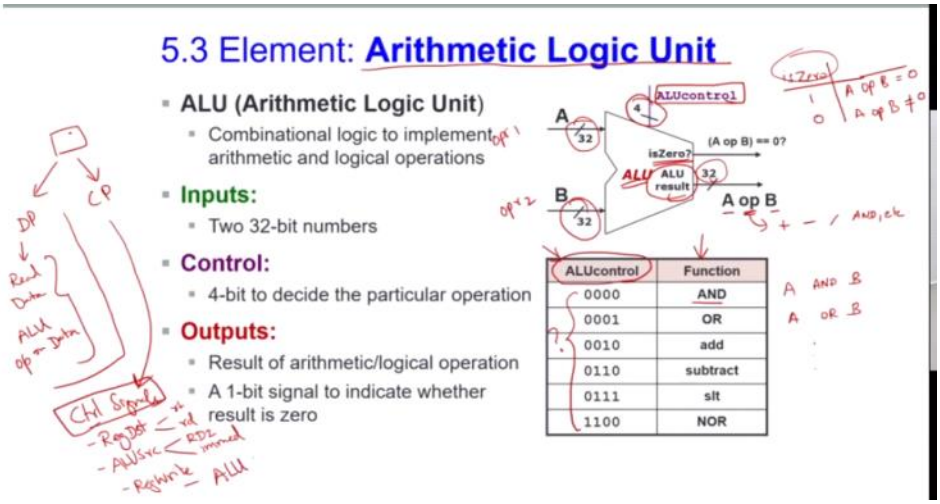


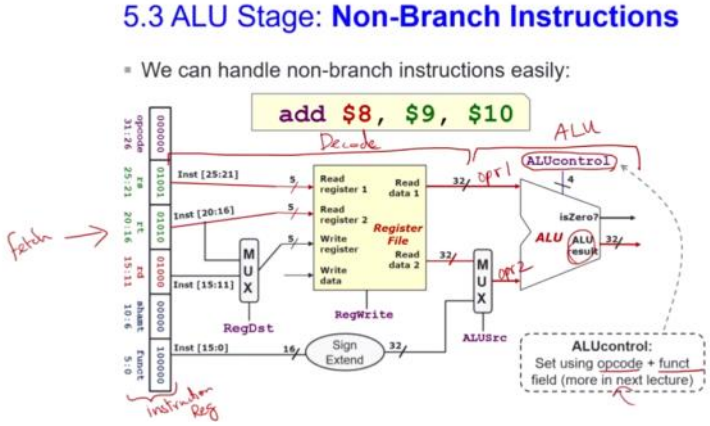



ALU

- Arithmetic logic unit
- Execution stage
- Does arimetic, shifting,logical ops
- For memory (lw, sw), address calc is done
- For branch ops, register comparison and target address calc is done

- 


- 


- 

## 5.3 ALU Stage: **Branch Instructions**

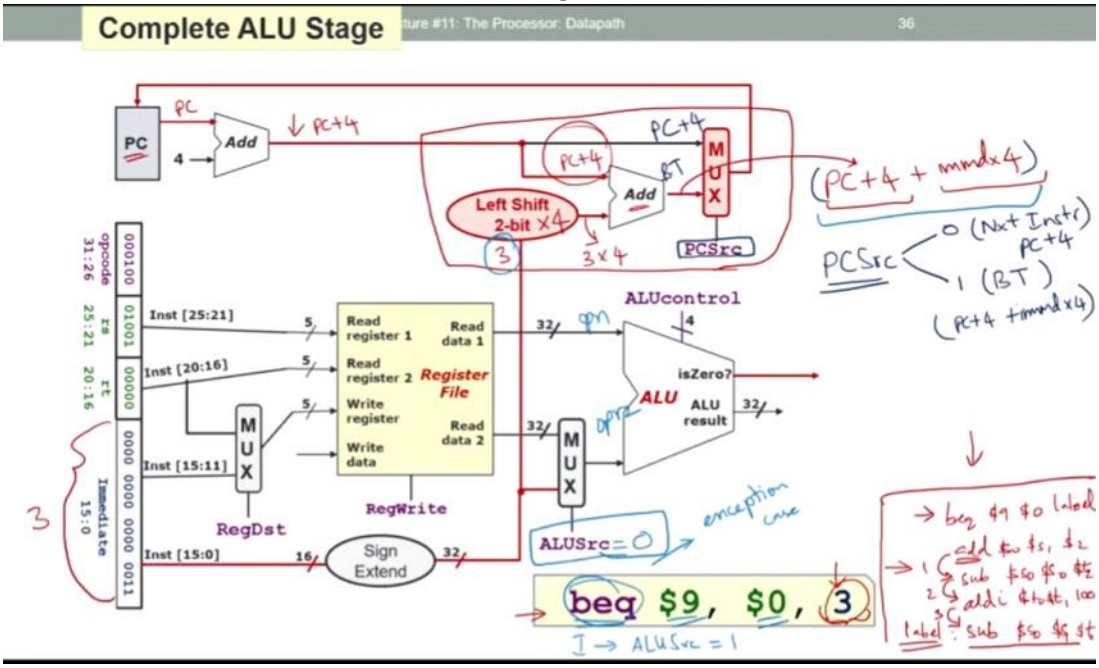
- Branch instruction is harder as we need to perform two calculations:
- Example: "`beq $9, $0, 3`"

**Note**
Two things need to happen to actually take the branch.
1. The instruction is a branch instruction.
2. The condition of the branch is true.

1. **Branch Outcome:**
   - Use ALU to compare the register
   - The 1-bit "`isZero?`" signal is enough to handle equal/not equal check (how?)
2. **Branch Target Address:**
   - Introduce additional logic to calculate the address
   - Need **PC** (from Fetch Stage)
   - Need **Offset** (from Decode Stage)

PC+4 + immed x 4

- Exception : Although branch instruction is I format, the ALUsrc = 1 usually, but here the ALUsrc = 0.
- New control signal: PCSrc can be 0, 1
  - To choose either PC + 4 (0) or Branch target address (1)

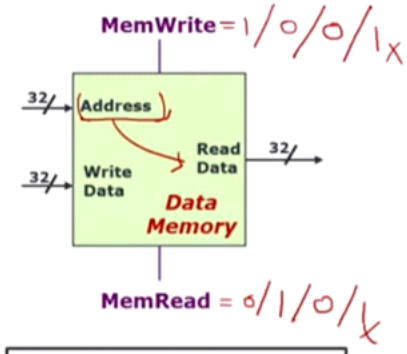



Memory Stage

- Only the load and store instructions perform operations in this stage
  - Use the memory address calculated by ALU
  - Read/write to data memory

# 5.4 Element: **Data Memory**

- Storage element for the data of a program
- **Inputs:**
  - Memory Address
  - Data to be written (Write Data) for store instructions
- **Control:**
  - Read and Write controls; only one can be asserted at any point of time
- **Output:**
  - Data read from memory (Read Data) for load instructions



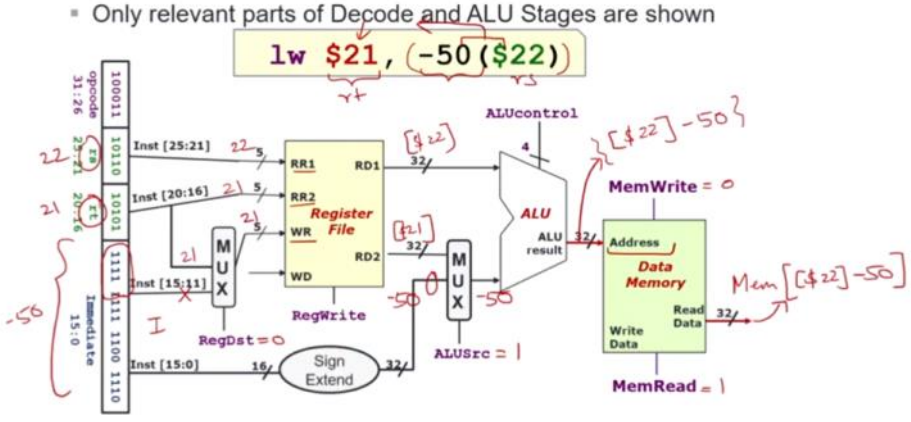

As a Function

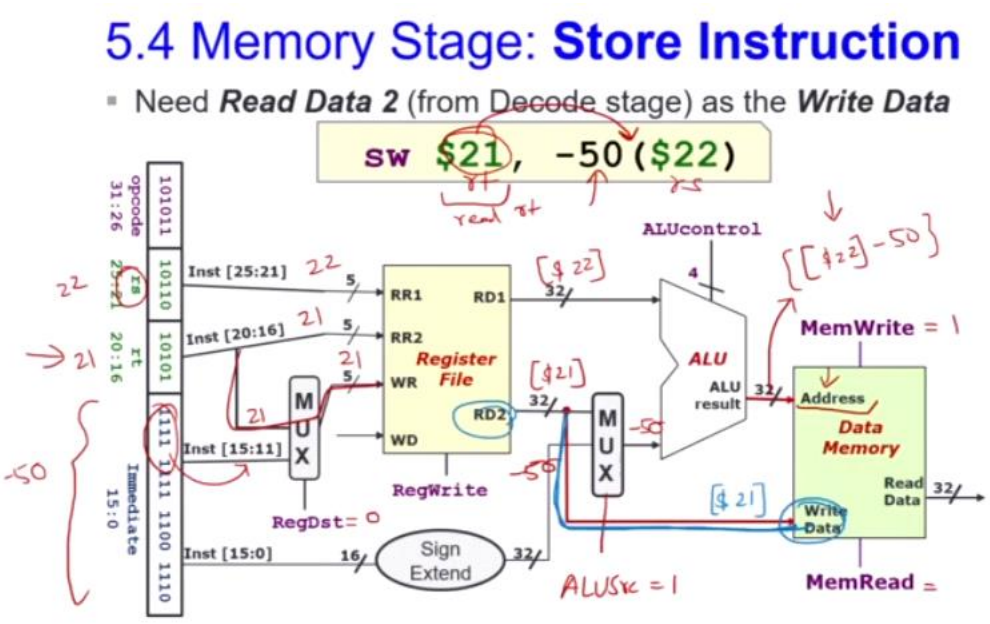
```
function DataMem(addr,WD,MW,MR) {
  if(MW) {
    Mem[addr] = WD;
  } else if(MR) {
    return Mem[addr];
  }
}
```

- Memwrite and memread cannot both be 1

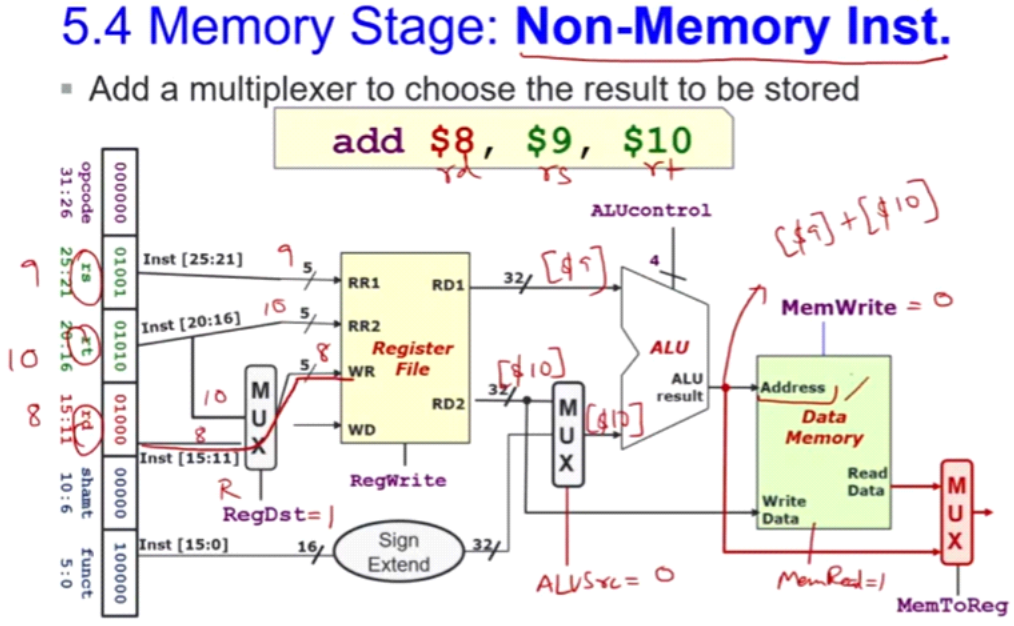
# 5.4 Memory Stage: **Load Instruction**

- Only relevant parts of Decode and ALU Stages are shown



- 


---

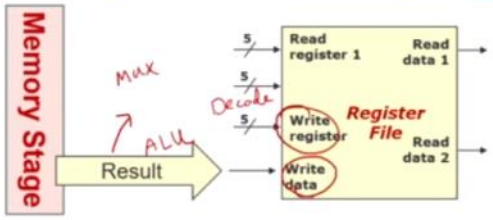
- 


---

Register Write Stage

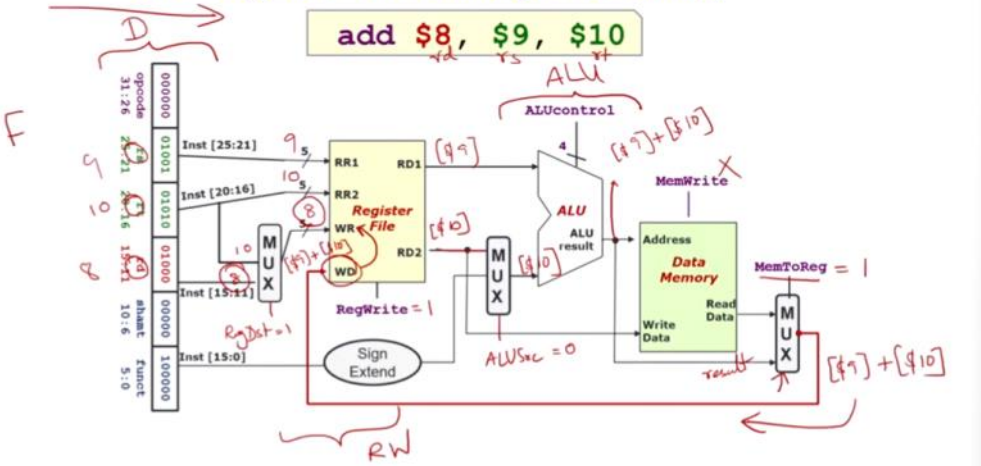
- Most instructions write result of computation into register
  - But not stores, branches, jumps

## 5.5 **Register Write Stage**: Block Diagram

- 


  - Result Write stage has no additional element:
    - Basically just route the correct result into register file
    - The ***Write Register*** number is generated way back in the **Decode** Stage

## 5.5 Register Write Stage: **Routing**

- 

# Week 6

Thursday, September 21, 2023 10:34 PM

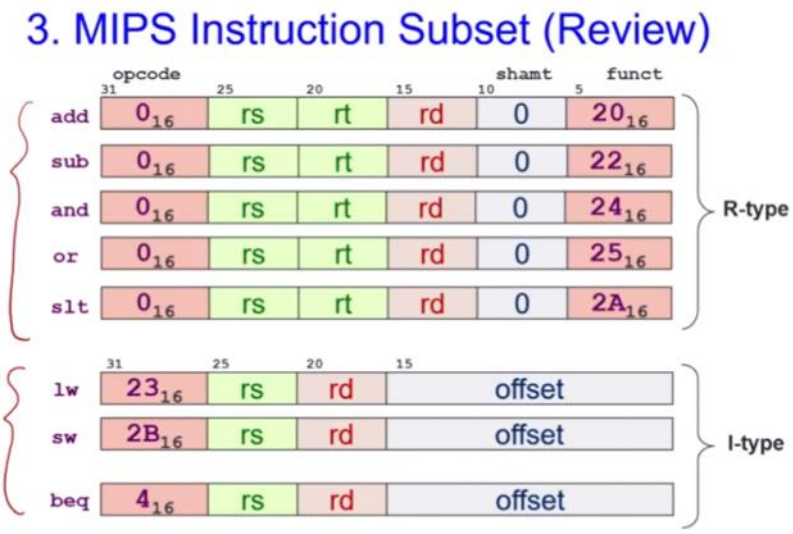
- Control signals
- RegDst, RegWrite,
- RegDst - choose rt or rd as destination register
    - If r format, rd is chosen
        Rd, rs, rt
    - If I format, rt is chosen
        - Rt, rs
- RegWrite - controls whether data from WD should be written to write register
- ALUSrc - choose right input, whether sign extended immediate operand or RD2
- ALUcontrol - performs different ops, like add,subtract
- MemWrite - control whether right data is write or read
- MemRead
- MemToReg - choose whether ALU result or read data
- PCSrc - if branch, then control where PC should go or simpl PC+4
- 1. Identified Control Signals

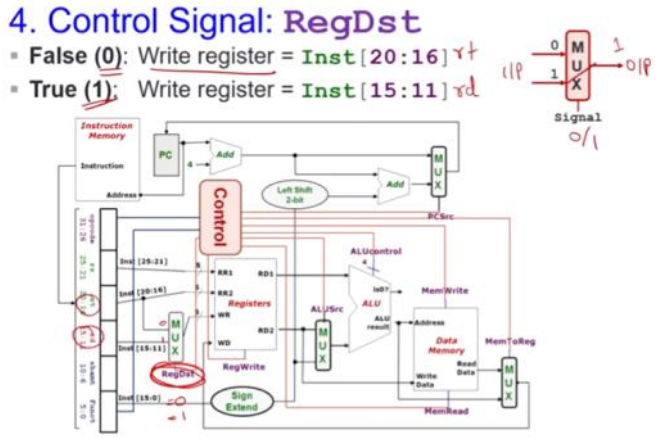
| Control Signal | Execution Stage | Purpose |
|---|---|---|
| RegDst | Decode/Operand Fetch | Select the destination register number |
| RegWrite | Decode/Operand Fetch RegWrite | Enable writing of register |
| ALUSrc | ALU | Select the 2nd operand for ALU |
| ALUcontrol | ALU | Select the operation to be performed |
| MemRead / MemWrite | Memory | Enable reading/writing of data memory |
| MemToReg | RegWrite | Select the result to be written back to register file |
| PCSrc | Memory/RegWrite | Select the next PC value |

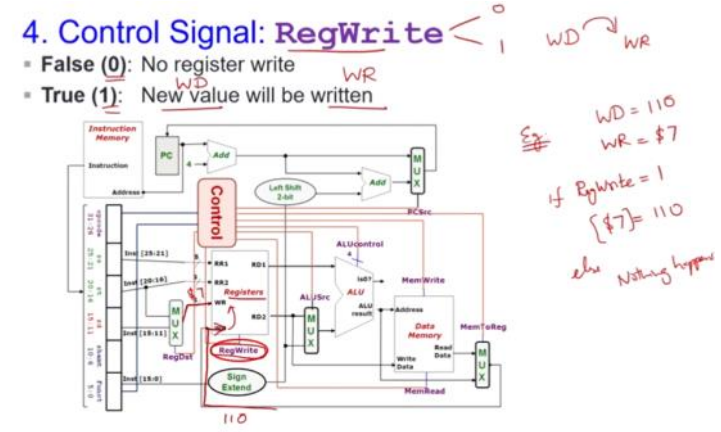
Generating the Control Signals

- Generated based on instruction to be executed
- All R-format have 0 opcode, J format have 2, and remaining are for I format
- R-format instruction
    - RegDst = 1
        - Because we choose the rd as destination register not rt
    - ALUSrc = 0
        - RD2 is passed to ALU as second operand, not a immediate
    - ALUcontrol
    - MemToReg = 0
    - Since r-type instruction has a 6 bit function code, this is used to generate the control signals
- A combinational circuit is a circuit where the output is immediately affected by the input
- A sequential circuit has a clock input, output changes only depending on behavior of clock signal and input
    - Ex Program counter
    - Only changes at rising clock

Implementing the control unit

- 

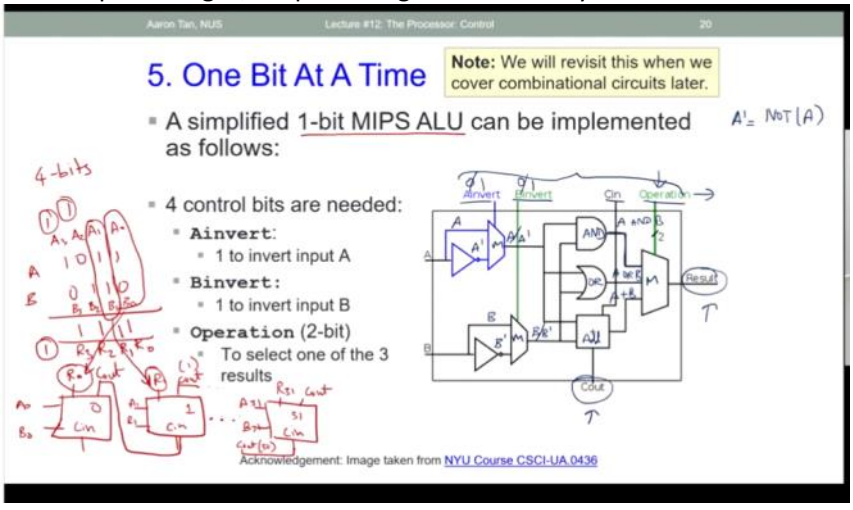

- RegDst
  - If control is 0, the rt taken
  - 

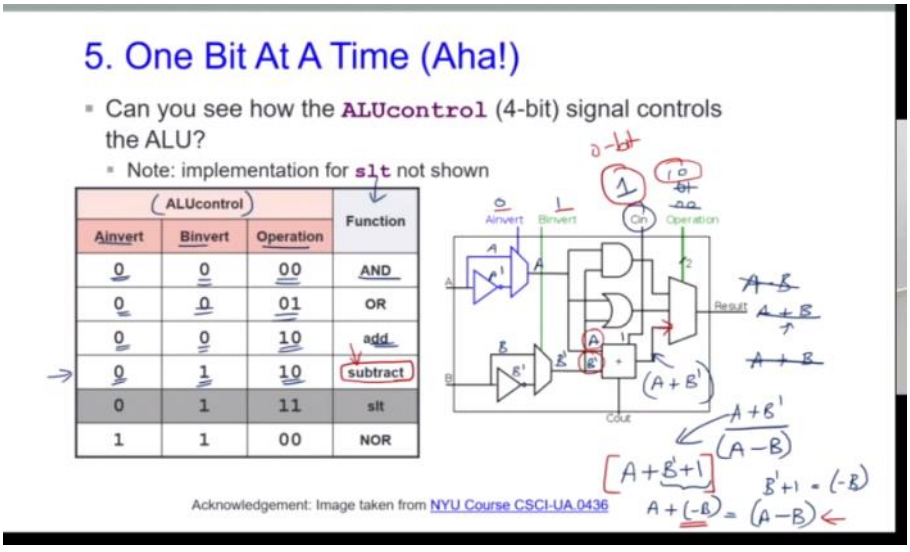
- RegWrite
  - If control is 0, nothing happens
  - If is 1, the data present in WD will be written to WR
  - 

- ALUSrc
  - Responsible to select the second operand
  - 1st operand comes from rs
  - Second chosen between rd2(rt) or sign extended immediate
- MemRead
  - If 0, then nothing read from memory

    - Otherwise, read from "address" and pass to "read data"
- MemWrite
    - If 0, then nothing written to memory
    - If 1, then whatever data is in "write data" is written to the "address"
- MemToReg
    - If 0, the ALU output written to register WD
    - If 1, then read data from memory is returned to WD
- PCSrc
    - Choose between PC+4 and Branch target address (PC + 4 + 4*immed)
    - If 1, branch should be taken
    - For beq, when the 2 register vals are same, the branch address is taken and PCSrc = 1
    - Register vals being same is given from ALU. Ex. Calc zero
    - Then updates the PC value
    - The opcode, tells if it's a branch instruction

ALUcontrol

- A 4 bit control signal
- When input changes, output changes immediately
- 

- To achieve A - B, ALU sets B invert to 1. to do A + B'. Cin set to one
    - So A + B' + 1 is eval (B' + 1) = -B
- 

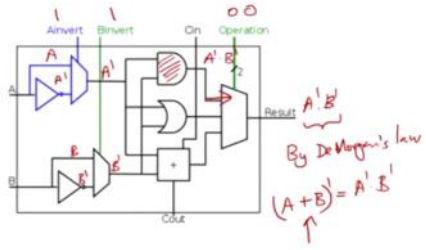

NOR

- Uses DeMorgans law, that (A and B)' = A' and B'

## 5. One Bit At A Time (Aha!)

- Can you see how the **ALUcontrol** (4-bit) signal controls the ALU?
  - Note: implementation for **slt** not shown

| 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 |




Acknowledgement: Image taken from NYU Course CSCI-UA.0436

Generating ALUcontrols signal

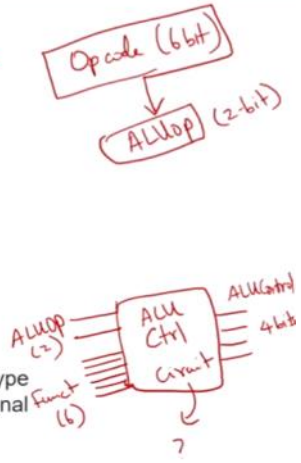
- They do depend on opcode(6bit) and function code (6 bit)
- We use a multilevel decoding approach
  - Use some of the input to reduce the cases, the generate full output
  - Done to reduce size of main controller
- INTERMEDIATE SIGNAL: ALUop
  - Use opcode to generate a 2bit ALUop signal
  - Represents whether lw,sw / bew / r-type
  - Use the ALUop signal and function code field (for R-type) to generate the 4 bit ALUcontrol signal

## 5. Intermediate Signal: ALUop

- **Basic Idea:**
  1. Use Opcode to generate a 2-bit **ALUop** signal
     - Represents classification of the instructions:

| Instruction type | ALUop |
|---|---|
| lw / sw | 00 |
| beq | 01 |
| R-type | 10 |

  2. Use **ALUop** signal and Function Code field (for R-type instructions) to generate the 4-bit **ALUcontrol** signal



- ## 5. Generating ALUcontrol Signal

| Opcode | ALUop | Instruction Operation | Funct field | ALU action | ALU control |
|---|---|---|---|---|---|
| lw | 00 | load word | xxxxxx | add | 0010 |
| sw | 00 | store word | xxxxxx | add | 0010 |
| beq | 01 | branch equal | xxxxxx | subtract | 0110 |
| R-type | 10 | add | 10 0000 | add | 0010 |
| R-type | 10 | subtract | 10 0010 | subtract | 0110 |
| R-type | 10 | AND | 10 0100 | AND | 0000 |
| R-type | 10 | OR | 10 0101 | OR | 0001 |
| R-type | 10 | set on less than | 10 1010 | set on less than | 0111 |

Generation of 2-bit ALUop signal will be discussed later

| Instruction Type | ALUop |
|---|---|
| lw / sw | 00 |
| beq | 01 |
| R-type | 10 |

| ALUcontrol | Function |
|---|---|
| 0000 | AND |
| 0001 | OR |
| 0010 | add |
| 0110 | subtract |
| 0111 | slt |
| 1100 | NOR |

- ## 5. Design of ALU Control Unit (1/2)
  - Input: 6-bit Funct field and 2-bit ALUop
  - Output: 4-bit ALUcontrol
  - Find the simplified expressions

| | 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 | 0010 |
| sw | 0 | 0 | X | X | X | X | X | X | 0010 |
| beq | 0 | 1 | X | X | X | X | X | X | 0110 |
| add | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0010 |
| sub | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0110 |
| and | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0000 |
| or | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0001 |
| slt | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0111 |

$A_3 =$
$A_2 =$
$A_1 =$
$A_0 =$
Circuit

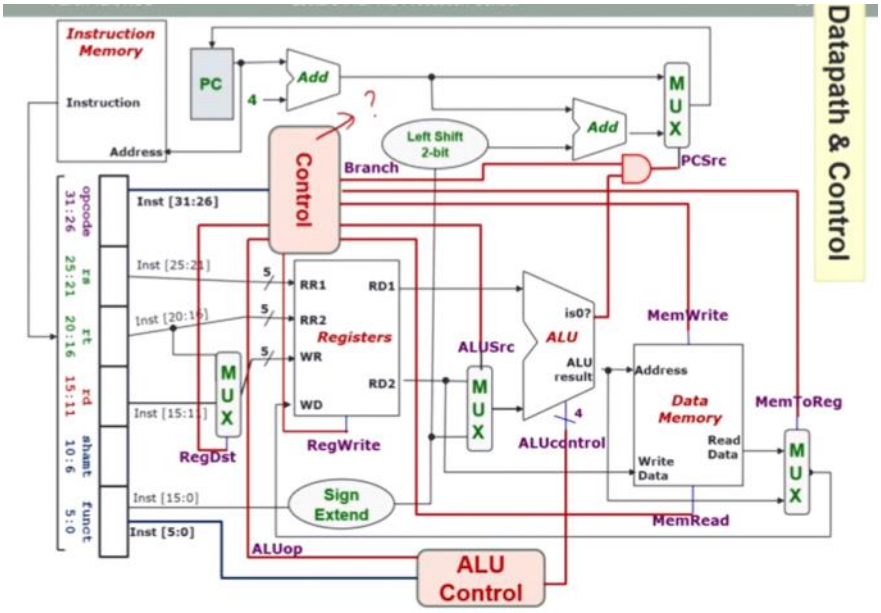
- ## 5. Design of ALU Control Unit (1/2)
  - Input: 6-bit Funct field and 2-bit ALUop
  - Output: 4-bit ALUcontrol
  - Find the simplified expressions

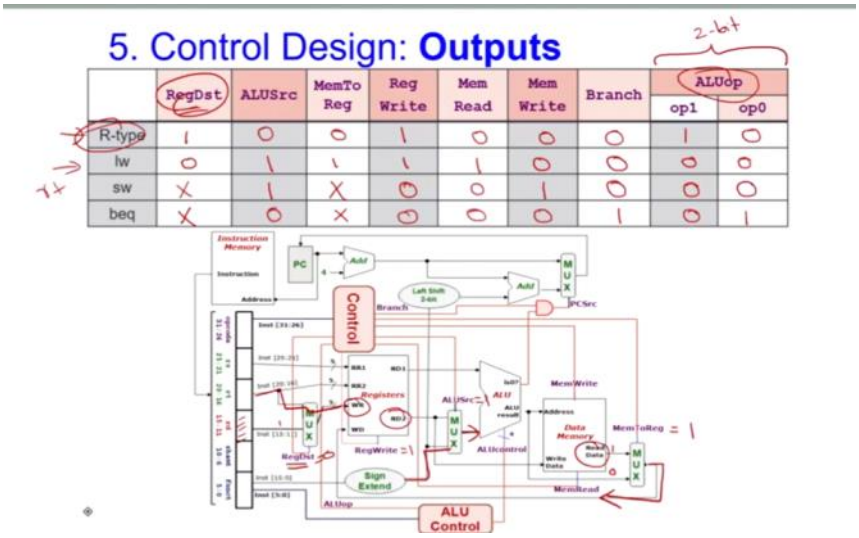
ALUcontrol3 = 0
ALUcontrol2 = ?
ALUop0 + ALUop1 · F1

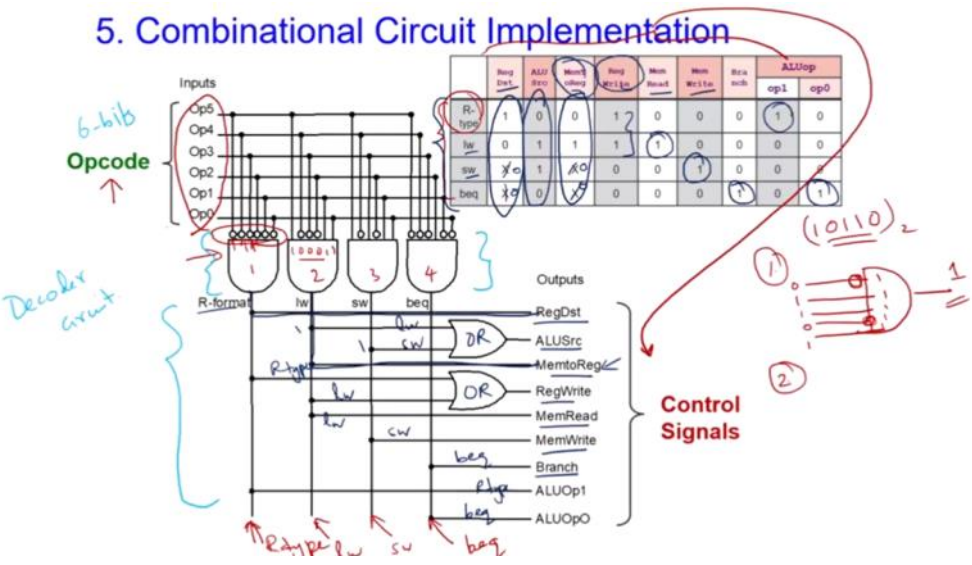
| | ALUop | | Funct Field ( F[5:0] == Inst[5:0] ) | | | | | | ALU control |
|---|---|---|---|---|---|---|---|---|---|
| | MSB | LSB | F5 | F4 | F3 | F2 | F1 | F0 | |
| lw | 0 | 0 | X | X | X | X | X | X | 0010 |
| sw | 0 | 0 | X | X | X | X | X | X | 0010 |
| beq | 0 | 1 | X | X | X | X | X | X | 0110 |
| add | 1 | X | 1 | X | 0 | 0 | 0 | 0 | 0010 |
| sub | 1 | X | 1 | X | 0 | 0 | 1 | 0 | 0110 |
| and | 1 | X | 1 | X | 0 | 1 | 0 | 0 | 0000 |
| or | 1 | X | 1 | X | 0 | 1 | 0 | 1 | 0001 |
| slt | 1 | X | 1 | X | 1 | 0 | 1 | 0 | 0111 |

$A_3 =$
$A_2 =$
$A_1 =$
$A_0 =$
Circuit

Control Design

- 

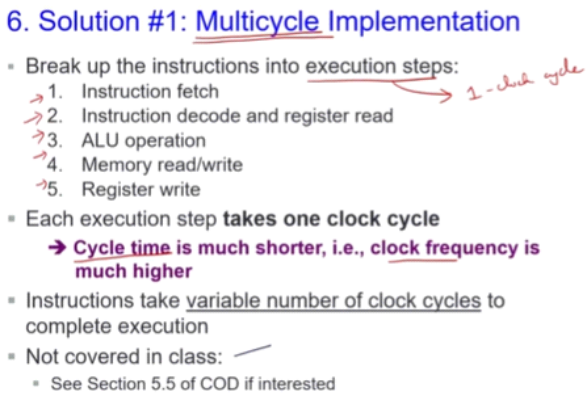
- 

- 


Instruction Execution

- Instructions are executed within a single clock cycle

- This has a drawback
- So instead, multicycle implementation is used




- Or use pipelining

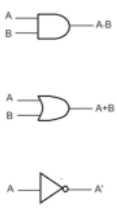
# Week 7

Thursday, October 12, 2023 11:44 AM

- Logic gates

## 2. Boolean Algebra

- Boolean values:
  - True (T or 1)
  - False (F or 0)
- Connectives
  - Conjunction (AND)
    - A · B; A ∧ B
  - Disjunction (OR)
    - A + B; A ∨ B
  - Negation (NOT)
    - A'; $\bar{A}$ ; ¬A;




## 4. Precedence of Operators

- Precedence from highest to lowest
  - Not (')
  - And (·)
  - Or (+)

  Note the difference with CS1231/CS1231S. Here in CS2100, AND has higher precedence than OR.
- Examples:
  - A · B + C = (A · B) + C  Hence, A · B + C is not ambiguous in CS2100.
  - X + Y' = X + (Y')
  - P + Q' · R = P + ((Q') · R)
- Use parenthesis to overwrite precedence. Examples:
  - A · (B + C) [ Without parenthesis, it means A·B+C or (A·B)+C ]
  - (P + Q)' · R [ Without parenthesis, it means P+Q'·R or P+(Q'·R) ]

## 5. Laws of Boolean Algebra

| | |
|---|---|
| **Identity laws** | |
| A + 0 = 0 + A = A | A · 1 = 1 · A = A |
| **Inverse/complement laws** | |
| A + A' = A' + A = 1 | A · A' = A' · A = 0 |
| **Commutative laws** | |
| A + B = B + A | A · B = B · A |
| **Associative laws *** | |
| A + (B + C) = (A + B) + C | A · (B · C) = (A · B) · C |
| **Distributive laws** | |
| A · (B + C) = (A · B) + (A · C) | A + (B · C) = (A + B) · (A + C) |

* Due to the associative laws, A + B + C is unambiguous. It may be evaluated as A + (B + C) or (A + B) + C. Likewise for A·B·C.

Duality

- If And/Or operaotrs and identity elements 0/1 in an equation are interchanged, it remains valid
- Ex. a+(b*c) = (a+B)*(a+c) is a*(b+c) = (a*b)+(a*c)
- If (x+y+z)' = x'*y'*z' is valid, then (x*y*z) = x' + y' + z' is also valid
- If x+1 = 1 is valid, then x*0 = 0i s valid

## 7. Theorems

| | |
|---|---|
| **Idempotency** | |
| X + X = X | X · X = X |
| **One element / Zero element** | |
| X + 1 = 1 + X = 1 | X · 0 = 0 · X = 0 |
| **Involution** | |
| (X')' = X | |
| **Absorption 1** | |
| X + X·Y = X | X·(X + Y) = X |
| **Absorption 2** | |
| X + X'·Y = X + Y | X·(X' + Y) = X·Y |
| **DeMorgans' (can be generalised to more than 2 variables)** | |
| (X + Y)' = X' · Y' | (X · Y)' = X' + Y' |
| **Consensus** | |
| X·Y + X'·Z + Y·Z = X·Y + X'·Z | (X+Y)·(X'+Z)·(Y+Z) = (X+Y)·(X'+Z) |

---

## 9. Complement Functions

- Given a Boolean function F, the complement of F, denoted as F', is obtained by interchanging 1 with 0 in the function's output values.
- Example: F1 = x·y·z'
- What is F1' ?
  - F1' = (x·y·z')'
    = x' + y' + (z')' (DeMorgan's)
    = x' + y' + z (Involution)

| x | y | z | F1 | F1' |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 |

Standard Forms

- Certain types of boolean expression lead to circuits that are desirable
- Sum-ofProducts (SOP)
- Product-of-sums (POS)
- Literals
  - Boolean variable on its own or is complement
  - x, x'
- Product term
  - Single literal or a logical product (AND) of several literals
  - x, x*y*z', A'*B
- Sum term
  - Single literal or logical sum (OR) of several literals
  - x, x+y+z', A'+B, A+B
- SOP
  - Product term or logical sum (OR) of several product terms
  - x, x+y*z'
    - A*B + A'*B'
    - No brackets needed
- POS
  - Sum term or a logical product (AND) of several sum terms
    - x*(y+z')
    - (A+B)*(A'+B')
- Every Boolean expression can be expression in SOP or POS

| | Expression | SOP? | POS? |
|---|---|---|---|
| (1) | X'·Y ⊕ X·Y'·⊕ X·Y·Z | ✓ | × |
| (2) | (X+Y')(X+Y)(X'+Z') | × | ✓ |
| (3) | X' + Y + Z' | ✓ | ✓ |
| (4) | X·(W' + Y·Z) | × | × |
| (5) | XYYZ | ✓ | ✓ |
| (6) | W·X'·Y + V·(X·Z + W') | × | × |

## 11. Minterms and Maxterms (1/2)

- A minterm of *n* variables is a product term that contains *n* literals from all the variables.
  - Example: On 2 variables x and y, the minterms are: x'·y', x'·y, x·y' and x·y
- A maxterm of *n* variables is a sum term that contains *n* literals from all the variables.
  - Example: On 2 variables x and y, the maxterms are: x'+y', x'+y, x+y' and x+y
- In general, with *n* variables we have up to $2^n$ minterms and $2^n$ maxterms.

## 11. Minterms and Maxterms (2/2)

- The minterms and maxterms on 2 variables are denoted by m0 to m3 and M0 to M3 respectively.

| 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 |

- Important fact: Each minterm is the complement of its corresponding maxterm. Likwise, each maxterm is the complement of its corresponding minterm.
  - Example: m2 = x·y'
    m2' = ( x·y' )' = x' + ( y' )' = x' + y = M2

## Quiz Time Again!

- Ability to convert minterms and maxterms from its Boolean expression to its notation (and vice versa) is important.
- Test yourself with the following quiz, assuming that you are given a Boolean function on 4 variables A, B, C, D.

Minterm

| | Boolean expression | Minterm notation |
|---|---|---|
| (1) | A'·B'·C·D | m3 |
| (2) | A·B'·C·D' | m10 |
| (3) | A·B'·C·D | m11 |
| (4) | A·B·C·D' | **m14** |
| (5) | A·B'·C'·D | **m9** |

Maxterm

| | Boolean expression | Maxterm notation |
|---|---|---|
| (1) | A+B+C'+D' | M3 |
| (2) | A'+B'+C+D' | M13 |
| (3) | A+B+C+D | M0 |
| (4) | A+B+C'+D | **M2** |
| (5) | A'+B+C+D' | **M9** |

## 12. Canonical Forms

- Canonical/normal form: a unique form of representation.
  - Sum-of-minterms = Canonical sum-of-products
  - Product-of-maxterms = Canonical product-of-sums

## 12.1 Sum-of-Minterms

- Given a truth table, example:

| x | y | z | F1 | F2 | F3 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 |

- Obtain sum-of-minterms expression by gathering the minterms of the function (where output is 1).

F1 = x·y·z' = m6

F2 = x'·y'·z + x·y'·z' + x·y'·z + x·y·z' + x·y·z
= m1 + m4 + m5 + m6 + m7 = Σm(1,4,5,6,7) or Σm(1,4 – 7)

F3 = x'·y'·z + x'·y·z + x·y'·z' + x·y'·z
= m1 + m3 + m4 + m5 = Σm(1,3,4,5) or Σm(1,3 – 5)

## 12.2 Product-of-Maxterms

- Given a truth table, example:

| x | y | z | F1 | F2 | F3 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 |

- Obtain product-of-maxterms expression by gathering the maxterms of the function (where output is 0).

F2 = (x+y+z) · (x+y'+z) · (x+y'+z')
= M0 · M2 · M3 = ΠM(0,2,3)

F3 = (x+y+z) · (x+y'+z) · (x'+y'+z) · (x'+y'+z')
= M0 · M2 · M6 · M7 = ΠM(0,2,6,7)

## 12.3 Conversion of Standard Forms

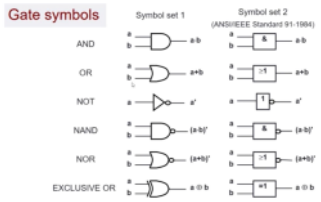
- We can convert between sum-of-minterms and product-of-maxterms easily
- Example: F2 = Σm(1,4,5,6,7) = ΠM(0,2,3)
- Why? See F2' in truth table.

| x | y | z | F2 | F2' |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 |

- F2' = m0 + m2 + m3
  Therefore,
  F2 = (m0 + m2 + m3)'
  = m0' · m2' · m3' (by DeMorgan's)
  = M0 · M2 · M3 (as mx' = Mx)

# 1. Logic Gates

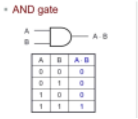
## Gate symbols




## 1.1 Inverter/AND/OR Gates

- Inverter (NOT gate)

- AND gate

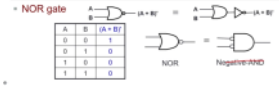


- OR gate

## 1.2 NAND/NOR Gates

- NAND gate

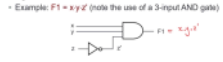

- NOR gate



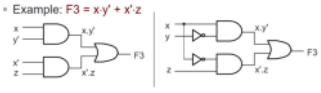

- XOR gate

- XNOR gate

Logic Circuits

- Fan-in : the number of inputs of a gate
- Every input must be connected
  - Example: F1 = x·y·z' (note the use of a 3-input AND gate)
  - 
  - Example: F2 = x + y'·z


  - Example: F3 = x·y' + x'·z

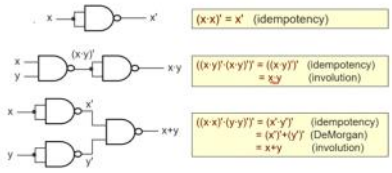



- Draw solid circle to denote wires internsect

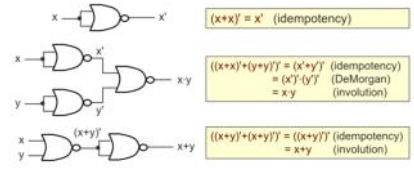
Universal Gates

- AND/OR/NOT gates are sufficient for building any function
- XOR gate for parity bit generation is useful
- {AND, OR, NOT} is complete set of logic

NAND Gate - universal

- NAND gate alone is complete set of logic
  - {NAND} is a complete set of logic.
  - Proof: Implement NOT/AND/OR using only NAND gates.
  - 

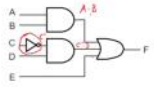

NOR Gate - universal

- {NOR} is complete set of logic
  - {NOR} is a complete set of logic.
  - Proof: Implement NOT/AND/OR using only NOR gates.
  - 

SOP and NAND circuits

## 3.3 SOP and NAND Circuits (1/2)

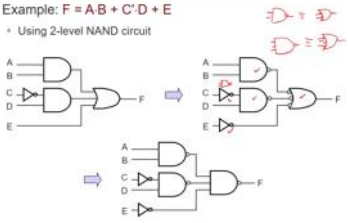
- An SOP expression can be easily implemented using
  - 2-level AND-OR circuit
  - 2-level NAND circuit
- Example: F = A·B + C'·D + E
  - Using 2-level AND-OR circuit




- Inverter is not considered a level

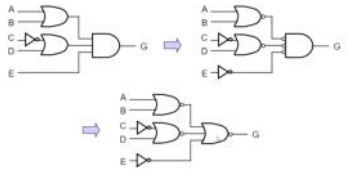
## 3.3 SOP and NAND Circuits (2/2)

- Example: F = A·B + C'·D + E
  - Using 2-level NAND circuit




## 3.4 POS and NOR Circuits (2/2)

- Example: G = (A+B) · (C'+D) · E
  - Using 2-level NOR circuit

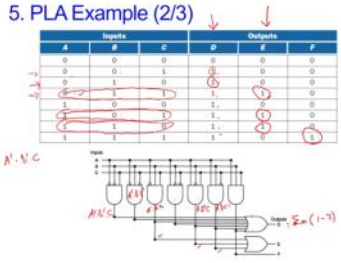



- Add bubble to inputs of and/or circuit to get the nor circuit

Integrated Circuit (IC) chip

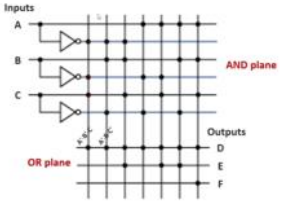
- Quad NAND gate

## 5. PLA Example (2/3)



| Inputs | | | Outputs | | |
|---|---|---|---|---|---|
| A | B | C | D | E | F |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 1 |






Function Simplification

Algebraic simplification

- Aim to minimize number of literals

## 2. Algebraic Simplification (2/3)

- Example 1: Simplify (x+y)·(x+y')·(x'+z)

| | |
|---|---|
| (x+y)·(x+y')·(x'+z) | |
| = (x·x + x·y' + x·y + y·y') · (x'+z) | (distributivity) |
| = (x + x·y' + x·y + y·y') · (x'+z) | (idempotency) |
| = (x + x·(y'+y) + y·y') · (x'+z) | (distributivity) |
| = (x + x·(1) + 0) · (x'+z) | (complement) |
| = (x + x) · (x'+z) | (identity) |
| = x · (x'+z) | (idempotency) |
| = x·x' + x·z | (distributivity) |
| = 0 + x·z | (complement) |
| = x·z | (identity) |

## 2. Algebraic Simplification (3/3)

- Example 2: Find the simplified SOP expression of
  F(a,b,c,d) = a·b·c + a·b·d + a'·b·c' + c·d + b·d'

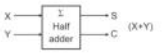
| | |
|---|---|
| a·b·c + a·b·d + a'·b·c' + c·d + b·d' | |
| = a·b·c + a·b + a'·b·c' + c·d + b·d' | (absorption 2) |
| = a·b·c + a·b + a'·b·c' + c·d + b·d' | (absorption 2) |
| = a·b + b·c' + c·d + b·d' | (absorption 1) |
| = a·b + c·d + b·(c'+d') | (distributivity) |
| = a·b + c·d + b·(c·d)' | (DeMorgan's) |
| = a·b + c·d + b | (absorption 2) |
| = b + c·d | (absorption 1) |

a·b·d + b·d'
= b·(a·d + d')
= b·(a + d')
= a·b + b·d'

Number of literals reduced from 13 to 3.

## 3. Half Adder (1/2)

- **Half adder** is a circuit that adds 2 single bits (X, Y) to produce a result of 2 bits (C, S).
- The black-box representation and truth table for half adder are shown below.




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

## 3. Half Adder (2/2)

- In canonical form (sum-of-minterms):
  - C = X·Y
  - S = X'·Y + X·Y'
- Output S can be simplified further (though no longer in SOP form):
  - S = X'·Y + X·Y' = X ⊕ Y
- Implementation of a half adder


Gray Code

- Unweighted, not arithmetic code
- Only a single bit change from 1 code value to next
- Not restricted to decimal nbit = 2^n val
- Good for error detection
- Reflected binary code

## 4. Gray Code (1/3)

- Unweighted (not an arithmetic code)
- Only a single bit change from one code value to the next.
  - Not restricted to decimal digits: *n* bits → $2^n$ values.
  - Good for error detection.
  - Named after Frank Gray; also called reflected binary code.
- Example: 4-bit standard Gray code

| Decimal | Binary | Gray Code | Decimal | Binary | Gray code |
|---|---|---|---|---|---|
| 0 | 0000 | 0000 | 8 | 1000 | 1100 |
| 1 | 0001 | 0001 | 9 | 1001 | 1101 |
| 2 | 0010 | 0011 | 10 | 1010 | 1111 |
| 3 | 0011 | 0010 | 11 | 1011 | 1110 |
| 4 | 0100 | 0110 | 12 | 1100 | 1010 |
| 5 | 0101 | 0111 | 13 | 1101 | 1011 |
| 6 | 0110 | 0101 | 14 | 1110 | 1001 |
| 7 | 0111 | 0100 | 15 | 1111 | 1000 |

- There are many grey code sequences

## 4. Gray Code (2/3)

- There are many Gray code sequences.
- Example: For 3 bits, here are some possible Gray code sequences:


## 4. Gray Code (3/3)

- Generating a 4-bit standard Gray code sequence.

- First 2 are same as binary, then draw a line and mirror it. Change next msb to 1. then do again after 4 numbers, then 8 numbers, then 16

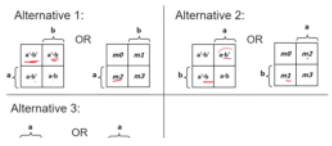
K-maps

- Systematic method to find simplified sum-of-products (SOP)
- Limited to 5/6 variables

- Karnaugh-map (K-map) is an abstract form of Venn diagram, organised as a matrix of squares, where
  - Each square represents a minterm
  - Two adjacent squares represent minterms that differ by exactly one literal

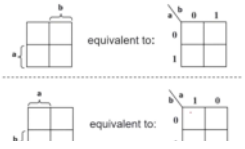
## 5.1 2-Variable K-maps (1/3)

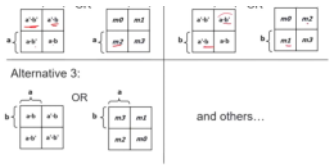
- Let the 2 variables be a and b. The K-map can be drawn as…
- Alternative layouts of a 2-variable (a, b) K-map:

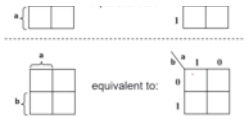



## 5.1 2-Variable K-maps (2/3)

- Alternative labelling of a 2-variable (a, b) K-map:

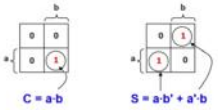




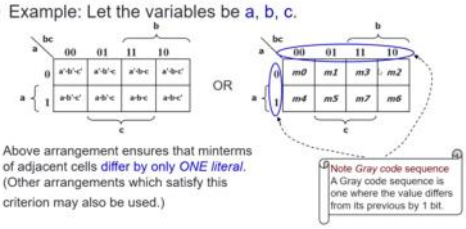

## 5.1 2-Variable K-maps (3/3)

- The K-map for a function is filled by putting
  - A '1' in the square the corresponds to a minterm of the function
  - A '0' otherwise
- Example: Half adder.




## 5.1 3-Variable K-maps (1/2)

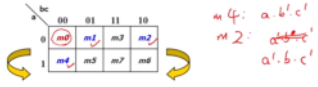
- As there are 8 minterms for 3 variables, so there are 8 squares in a 3-variable K-map.
- Example: Let the variables be a, b, c.




Above arrangement ensures that minterms of adjacent cells differ by only *ONE literal*. (Other arrangements which satisfy this criterion may also be used.)

## 5.1 3-Variable K-maps (2/2)

- There is wrap-around in the K-map:
  - a'·b'·c' (*m0*) is adjacent to a'·b·c' (*m2*)
  - a·b'·c' (*m4*) is adjacent to a·b·c' (*m6*)



Each cell in a 3-variable K-map has 3 adjacent neighbours. For example, *m0* has 3 adjacent neighbours: *m1*, *m2* and *m4*.

In general, each cell in an *n*-variable K-map has *n* adjacent neighbours.

### Quick Review Questions #1

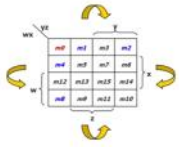
- DLD page 106, questions 5-1 to 5-2.

5-1. The K-map of a 3-variable function *F* is shown below. What is the sum-of-minterms expression of *F*?

5-2. Draw the K-map for this function *A*:
*A*(x, y, z) = x·y + y·z' + x'·y'·z

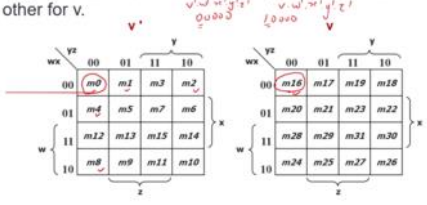
## 5.1 4-Variable K-maps (2/2)

- There are 2 wrap-arounds.
- Every cell has 4 neighbours.
  - Example: The cell corresponding to minterm *m0* has neighbours *m1*, *m2*, *m4* and *m8*.



## 5.1 5-Variable K-maps (2/2)

- Organised as two 4-variable K-maps. One for v' and the other for v.



## 5.2 How to Use K-maps (1/7)

- Based on the Unifying Theorem (complement law):

  **A + A' = 1**

- In a K-map, each cell containing a '1' corresponds to a minterm of a given function *F* where the output is 1.
- Each valid grouping of adjacent cells containing '1' then corresponds to a simpler product term of *F*.
  - A group must have size in powers of two: 1, 2, 4, 8, …
  - Grouping 2 adjacent cells eliminates 1 variable from the product term; grouping 4 cells eliminates 2 variables; grouping 8 cells eliminates 3 variables, and so on. In general, grouping $2^n$ cells eliminates *n* variables.

## 5.2 How to Use K-maps (2/7)

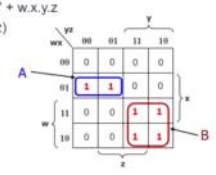
- Group as many cells as possible
  - The larger the group, the fewer the number of literals in the resulting product term.
- Select as few groups as possible to cover all the cells (minterms) of the function
  - The fewer the groups, the fewer is the number of product terms in the simplified SOP expression.

## 5.2 How to Use K-maps (4/7)

- Each group of adjacent minterms corresponds to a possible product term of the given function.
- Here, there are 2 groups of minterms, A and B:

  A = w'.x.y'.z' + w'.x.y'.z = w'.x.y'.(z' + z) = **w'.x.y'**

  B = w.x'.y.z' + w.x'.y.z + w.x.y.z' + w.x.y.z
  = w.x'.y.(z' + z) + w.x.y.(z' + z)
  = w.x'.y + w.x.y
  = w.(x'+x).y
  = **w.y**




## 5.2 How to Use K-maps (5/7)

- Each product term that corresponds to a group, w'·x·y' and w·y, represents the sum of minterms in that group.
- Boolean expression is therefore the sum of product terms (SOP) that represent all groups of the minterms of the function:

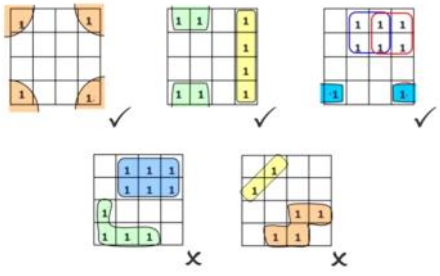
  *F*(w,x,y,z) = group A + group B = w'·x·y' + w·y

## 5.2 How to Use K-maps (6/7)

- The larger the group (the more minterms it contains), the fewer is the number of literals in the associated product term.
  - Recall that a group must have size in powers of two.
  - Example: For a 4-variable K-map with variables w, x, y, z
    1 cell = 4 literals. Examples: w·x·y·z, w'·x·y'·z
    2 cells = 3 literals. Examples: w·x·y, w·y'·z'
    4 cells = 2 literals. Examples: w·x, x'·y
    8 cells = 1 literal. Examples: w, y', z
    16 cells = no literal (i.e. logical constant 1). Example: 1

## 5.2 How to Use K-maps (7/7)

- Examples of valid and invalid groupings.



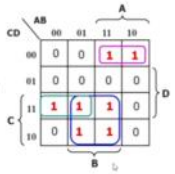
## 5.3 Converting to Minterms Form (2/2)

- Example:

  F(A,B,C,D) = A.(C+D)'.(B'+D') + C.(B+C'+A'.D)
  = A.(C'.D').(B'+D') + B.C + C.C' + A'.C.D
  = A.B'.C'.D' + A.C'.D' + B.C + A'.C.D

Expanding it to sum of minterms (unnecessary):

A.B'.C'.D' + A.C'.D' + B.C + A'.C.D
= A.B'.C'.D' + A.C'.D'.(B+B') + B.C + A'.C.D
= A.B'.C'.D' + A.B.C'.D' + A.B'.C'.D' + B.C.(A+A') + A'.C.D
= A.B'.C'.D' + A.B.C'.D' + A.B.C + A'.B.C + A'.C.D
= A.B'.C'.D' + A.B.C'.D' + A.B.C.(D+D') + A'.B.C.(D+D') + A'.C.D.(B+B')
= A.B'.C'.D' + A.B.C'.D' + A.B.C.D + A.B.C.D' + A'.B.C.D + A'.B.C.D' + A'.B'.C.D



## 5.4 PIs and EPIs (1/3)

- To find the simplest (minimal) SOP expression from a K-map, you need to obtain:
  - Minimum number of literals per product term; and
  - Minimum number of product terms.
- Achieved through K-map using
  - *Bigger groupings* of minterms (prime implicants) where possible; and
  - *No redundant groupings* (look for essential prime implicants)
- Implicant: a product term that could be used to cover minterms of the function.

## 5.4 PIs and EPIs (2/3)

- Prime implicant (PI): a product term obtained by combining the *maximum possible number of minterms* from *adjacent* squares in the map. (That is, it is the biggest grouping possible.)
- Always look for prime implicants in a K-map.

## 5.4 PIs and EPIs (3/3)

- No redundant groups:

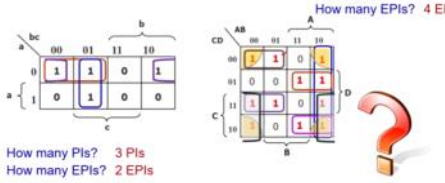
- Essential prime implicant (EPI): a prime implicant that includes at least one minterm that is not covered by any other prime implicant.

## Quick Review Questions #2

- DLD page 106, question 5-3.

5-3. Identify the prime implicants and essential prime implicants of the two K-maps below.

How many PIs? 7 PIs
How many EPIs? 4 EPIs



How many PIs? 3 PIs
How many EPIs? 2 EPIs

Finding simplified SOP exp

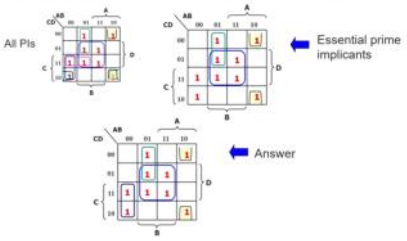
## 5.5 Finding Simplified SOP Expression (1/4)

- Algorithm
  1. Circle all prime implicants on the K-map.
  2. Identify and select all essential prime implicants for the cover.
  3. Select a minimum subset of the remaining prime implicants to complete the cover, that is, to cover those minterms not covered by the essential prime implicants.

## 5.5 Finding Simplified SOP Expression (3/4)

One additional prime implicant needed

= B*D + A'*B*C' + A*B'*D' + A'*B'*C

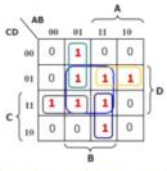



- DLD pages 106-107, questions 5-4 to 5-7.

5-4. Find the minimal SOP expression for G(A,B,C,D).

Prime implicants?
B·D, A'·B·C', A·C'·D, A·B·C, A'·C·D

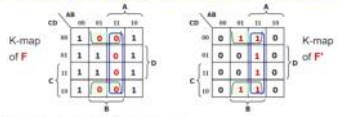
Essential prime implicants?
All, except B·D, are essential.



G = A'·B·C' + A·C'·D + A·B·C + A'·C·D

Simplified POS Expression

## 5.6 Finding Simplified POS Expression (2/2)




- This gives the SOP of F' to be
  F' = B·D' + A·B
- To get POS of F, we have

F = (B·D' + A·B)'
= (B·D')' · (A·B)' (DeMorgan)
= **(B'+D) · (A'+B')** (DeMorgan)

To go from SOP to POS, take the complement

## 5.7 Don't-Care Conditions (1/5)

- In certain problems, some outputs are not specified or are invalid. Hence, these outputs can be either '1' or '0'.
- They are called don't-care conditions, denoted by X (or d).
- Example: A circuit takes in a 3-bit value ABC and outputs 2-bit value FG which is the sum of the input bits. It is also known that inputs 000 and 111 never occur.

Assuming all inputs are valid.

| A | B | C | F | G |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

Assuming inputs 000 and 111 are invalid.

| A | B | C | F | G |
|---|---|---|---|---|
| 0 | 0 | 0 | X | X |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | X | X |

## 5.7 Don't-Care Conditions (2/5)

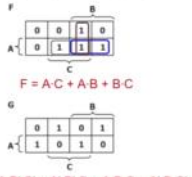
- Don't-care conditions can be used to help simplify Boolean expression further in K-maps.
- They could be chosen to be either '1' or '0', depending on which choice results in a simpler expression.
- We usually use the notation Σd to denote the set of don't-care minterms.
  - Example:
    F(A,B,C) = Σm(3, 5, 6) + Σd(0, 7)
    G(A,B,C) = Σm(1, 2, 4) + Σd(0, 7)

Assuming inputs 000 and 111 are invalid.

| A | B | C | F | G |
|---|---|---|---|---|
| 0 | 0 | 0 | X | X |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | X | X |

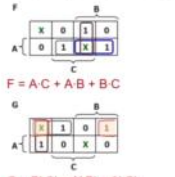
## 5.7 Don't-Care Conditions (3/5)

- Comparison
  - Without don't-cares:
    F(A,B,C) = Σm(3, 5, 6, 7)
    G(A,B,C) = Σm(1, 2, 4, 7)



F = A·C + A·B + B·C
G = A·B'·C' + A'·B'·C + A·B·C + A'·B·C'

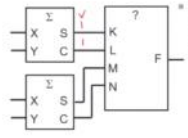
  - With don't-cares:
    F(A,B,C) = Σm(3, 5, 6) + Σd(0, 7)
    G (A,B,C)= Σm(1, 2, 4) + Σd(0, 7)



F = A·C + A·B + B·C
G = B'·C' + A'·B' + A'·C'

## 5.7 Don't-Care Conditions (4/5)

- Suppose you are given the truth table for a function F(K,L,M,N) as follows:
- You are also told that the inputs K, L, M, N are taken from the outputs of two half adders as shown:

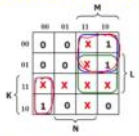



- Then you may revise the truth table:

| K | L | M | N | F |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | X |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | X |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | X |
| 1 | 1 | 0 | 0 | X |
| 1 | 1 | 0 | 1 | X |
| 1 | 1 | 1 | 0 | X |
| 1 | 1 | 1 | 1 | X |

## 5.7 Don't-Care Conditions (5/5)

- K-map of *F*:

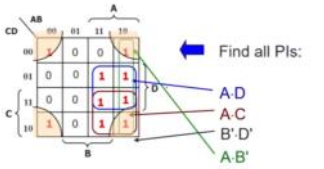


- PIs: K'·M, L·M, and K·M'·N'
  (Note: K·L and M·N not considered PIs as they consist of only X's.)
- EPIs: K'·M and K·M'·N'

| K | L | M | N | F |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | X |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | X |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | X |
| 1 | 1 | 0 | 0 | X |
| 1 | 1 | 0 | 1 | X |
| 1 | 1 | 1 | 0 | X |
| 1 | 1 | 1 | 1 | X |

## 6. More Examples (2/6)

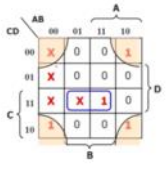
- Example #2:
  F(A,B,C,D) = A·B·C + B'·C·D' + A·D + B'·C'·D'




A.D, A.C and B'.D' are EPIs, and they cover all the minterms.
So the answer is: **F(A,B,C,D) = A·D + A·C + B'·D'**

## 6. More Examples (4/6)

- Example #3 (with don't-cares):
  F(A,B,C,D) = Σm(2,8,10,15) + Σd(0,1,3,7)



Do we need to have an additional term A'·B' to cover the 2 remaining X's?

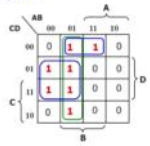
No, because all the 1's (minterms) have been covered.

Answer: **F(A,B,C,D) = B'·D' + B·C·D**

## 6. More Examples (5/6)

- Find the simplest POS expression for example #2:
  F(A,B,C,D) = A·B·C + B'·C·D' + A·D + B'·C'·D'
- Draw the K-map of the complement of F, that is, F'.

K-map of F'




From K-map,

F' = A'·B + A'·D + B·C'·D'

Using DeMorgan's theorem,

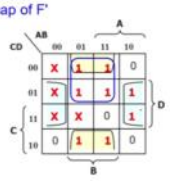
F = (A'·B + A'·D + B·C'·D')'

= **(A+B')·(A+D')·(B'+C+D)**

## 6. More Examples (6/6)

- Find the simplest POS expression for example #3:
  F(A,B,C,D) = Σm(2,8,10,15) + Σd(0,1,3,7)
- Draw the K-map of the complement of F, that is, F'.
  F'(A,B,C,D) = Σm(4,5,6,9,11,12,13,14) + Σd(0,1,3,7)

K-map of F'




From K-map,

F ' = B·C' + B·D' + B'·D

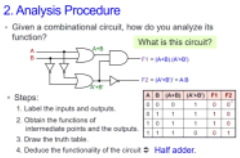
Using DeMorgan's theorem,

F = (B·C' + B·D' + B'·D)'

= **(B'+C)·(B'+D)·(B+D')**

# Week 8

Tuesday, October 17, 2023 2:27 AM




## 3. Design Methods

- Different combinational circuit design methods:
  - Gate-level design method (with logic gates)
  - Block-level design method (with functional blocks)
- Design methods make use of logic gates and useful function blocks
  - These are available as Integrated Circuit (IC) chips.
  - Types of IC chips (based on packing density): SSI, MSI, LSI, VLSI, ULSI.
- Main objectives of circuit design:
  - Reduce cost (number of gates for small circuits; number of IC packages for complex circuits)
  - Increase speed
  - Design simplicity (re-use blocks where possible)

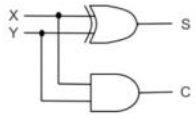
## 4. Gate-Level (SSI) Design: Half Adder (2/2)

4. Obtain simplified Boolean functions.
   Example: C = X·Y
   S = X'·Y + X·Y' = X⊕Y

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

5. Draw the logic diagram.

Half Adder




## 4. Gate-Level (SSI) Design: Full Adder (1/5)

- Half adder adds up only two bits.
- To add two binary numbers, we need to add 3 bits (including the *carry*).
  - Example:

| | | | | |
|---|---|---|---|---|
| | 1 | 1 | 1 | carry |
| | 0 | 0 | 1 | 1 | X |
| + | 0 | 1 | 1 | 1 | Y |
| | 1 | 0 | 1 | 0 | S |

## 4. Gate-Level (SSI) Design: Full Adder (2/5)

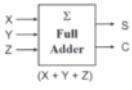
- Truth table:

| X | Y | Z | C | S |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

Note:
Z - carry in (to the current position)
C - carry out (to the next position)

C

| X \ YZ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 1 |




- Using K-map, simplified SOP form:

  C = X·Y + X·Z + Y·Z
  S = X'·Y'·Z + X'·Y·Z' + X·Y'·Z' + X·Y·Z

S

| X \ YZ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |

## 4. Gate-Level (SSI) Design: Full Adder (3/5)

- Alternative formulae using algebraic manipulation:

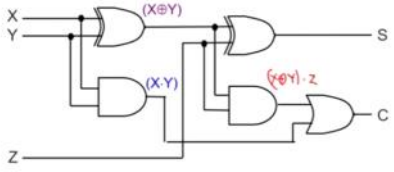
C = X·Y + X·Z + Y·Z
= X·Y + (X + Y)·Z
= X·Y + ( (X⊕Y) + X·Y )·Z
= X·Y + (X⊕Y)·Z + X·Y·Z
= X·Y + (X⊕Y)·Z

S = X'·Y'·Z + X'·Y·Z' + X·Y'·Z' + X·Y·Z
= X'·(Y'·Z + Y·Z') + X·(Y'·Z' + Y·Z)
= X'·(Y⊕Z) + X·(Y⊕Z)'
= X⊕(Y⊕Z)

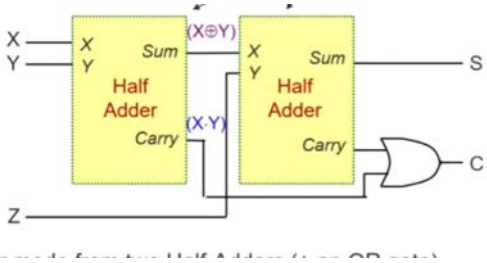
## 4. Gate-Level (SSI) Design: Full Adder (4/5)

- Circuit for above formulae:

C = X·Y + (X⊕Y)·Z
S = X⊕(Y⊕Z) = (X⊕Y)⊕Z (XOR is associative)




- Full adder made from 2 half adders + or gate




## 4. BCD to Excess-3 Code Converter (1/3)

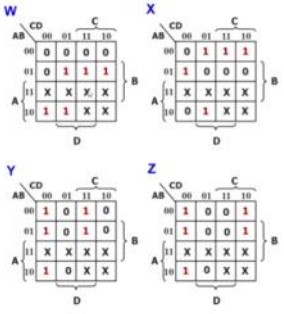
| Digit | BCD code | Excess-3 code |
|---|---|---|
| 0 | 0000 | 0011 |
| 1 | 0001 | 0100 |
| 2 | 0010 | 0101 |
| 3 | 0011 | 0110 |
| 4 | 0100 | 0111 |
| 5 | 0101 | 1000 |
| 6 | 0110 | 1001 |
| 7 | 0111 | 1010 |
| 8 | 1000 | 1011 |
| 9 | 1001 | 1100 |

## 4. BCD to Excess-3 Code Converter (2/3)

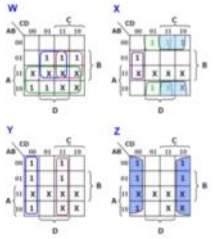
- Truth table:

| | BCD | | | | Excess-3 | | | |
|---|---|---|---|---|---|---|---|---|
| | A | B | C | D | W | X | Y | Z |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 3 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 4 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 5 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 6 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 7 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| 8 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 9 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 10 | 1 | 0 | 1 | 0 | X | X | X | X |
| 11 | 1 | 0 | 1 | 1 | X | X | X | X |
| 12 | 1 | 1 | 0 | 0 | X | X | X | X |
| 13 | 1 | 1 | 0 | 1 | X | X | X | X |
| 14 | 1 | 1 | 1 | 0 | X | X | X | X |
| 15 | 1 | 1 | 1 | 1 | X | X | X | X |

- K-maps:



## 4. BCD to Excess-3 Code Converter (3/3)



$W = A + B{\cdot}C + B{\cdot}D$

$X = B'{\cdot}C + B'{\cdot}D + B{\cdot}C'{\cdot}D'$

$Y = C{\cdot}D + C'{\cdot}D'$

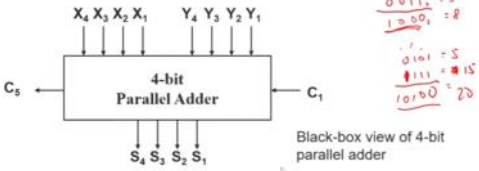
$Z = D'$

## 5. Block-Level Design

- More complex circuits can also be built using block-level method.
- In general, block-level design method (as opposed to gate-level design) relies on algorithms or formulae of the circuit, which are obtained by decomposing the main problem to sub-problems recursively (until small enough to be directly solved by blocks of circuits).
- First example shows how to create a 4-bit parallel adder using block-level design.
- Using 4-bit parallel adders as building blocks, we can create the following:
  1. BCD-to-Excess-3 Code Converter
  2. 16-bit Parallel Adder

## 5. 4-bit Parallel Adder (1/4)

- Consider a circuit to add two 4-bit numbers together and a carry-in, to produce a 5-bit result.

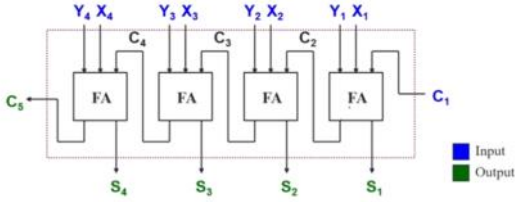



Black-box view of 4-bit parallel adder

- 5-bit result is sufficient because the largest result is:
  $1111_2 + 1111_2 + 1_2 = 11111_2$

## 5. 4-bit Parallel Adder (4/4)

- Cascading 4 full adders via their carries, we get:




- Note that carry is propagated by cascading the carry from one full adder to the next.
- Called Parallel Adder because inputs are presented simultaneously (in parallel). Also called Ripple-Carry Adder.

## 5. BCD to Excess-3 Converter: Revisit (1/2)

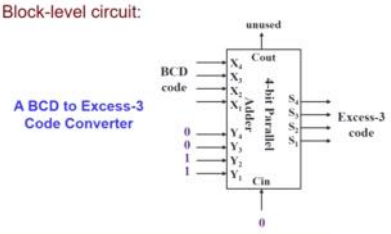
- Excess-3 code can be converted from BCD code using truth table:
- Gate-level design can be used since only 4 inputs.
- However, alternative design is possible.
- Use problem-specific formula:

  Excess-3 code = BCD Code + $0011_2$

| | BCD | | | | Excess-3 | | | |
|---|---|---|---|---|---|---|---|---|
| | A | B | C | D | W | X | Y | Z |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 3 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 4 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 5 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 6 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 7 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| 8 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 9 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 10 | 1 | 0 | 1 | 0 | X | X | X | X |
| 11 | 1 | 0 | 1 | 1 | X | X | X | X |
| 12 | 1 | 1 | 0 | 0 | X | X | X | X |
| 13 | 1 | 1 | 0 | 1 | X | X | X | X |
| 14 | 1 | 1 | 1 | 0 | X | X | X | X |
| 15 | 1 | 1 | 1 | 1 | X | X | X | X |

## 5. BCD to Excess-3 Converter: Revisit (2/2)

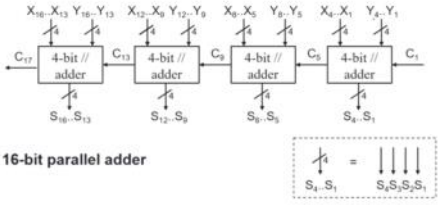
- Block-level circuit:




Note: In the lab, input 0 (low) is connected to GND, 1 (high) to Vcc.

## 5. 16-bit Parallel Adder

- Larger parallel adders can be built from smaller ones.
- Example: A 16-bit parallel adder can be constructed from four 4-bit parallel adders:

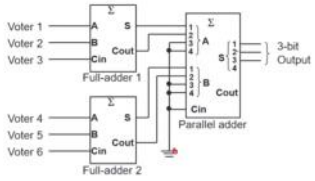



**A 16-bit parallel adder**



## 7. Example: 6-Person Voting System

- Application: 6-person voting system.
  - Use FAs and a 4-bit parallel adder.
  - Each FA can sum up to 3 votes.




## 8. Magnitude Comparator (1/4)

- Magnitude comparator: compares 2 unsigned values A and B, to check if A>B, A=B, or A<B.
- To design an *n*-bit magnitude comparator using classical method, it would require $2^{2n}$ rows in truth table!
- We shall exploit regularity in our design.
- Question: How do we compare two 4-bit unsigned values A ($a_3a_2a_1a_0$) and B ($b_3b_2b_1b_0$)?

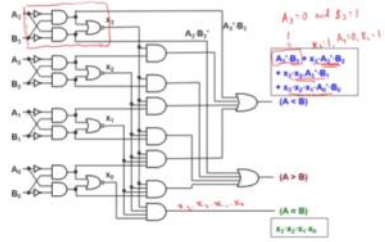
  If ($a_3 > b_3$) then A > B

  If ($a_3 < b_3$) then A < B

  If ($a_3 = b_3$) then if ($a_2 > b_2$) ...

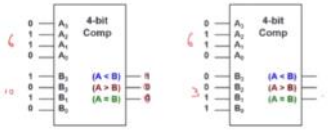
## 8. Magnitude Comparator (2/4)

Let $A = A_3A_2A_1A_0$, $B = B_3B_2B_1B_0$; $x_i = A_iB_i + A_i'B_i'$



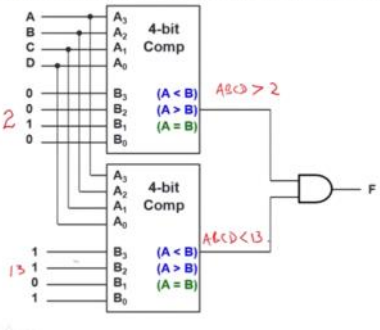

## 8. Magnitude Comparator (3/4)

- Block diagram of a 4-bit magnitude comparator




## 8. Magnitude Comparator (4/4)

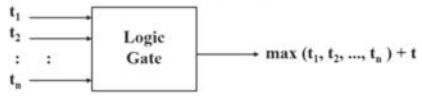
- A function F accepts a 4-bit binary value ABCD, and returns 1 if $3 \le ABCD \le 12$, or 0 otherwise. How would you implement F using magnitude comparators and a suitable logic gate?



## 9. Circuit Delays (1/5)

- Given a logic gate with delay t. If inputs are stable at times $t_1, t_2, \ldots, t_n$, then the earliest time in which the output will be stable is:

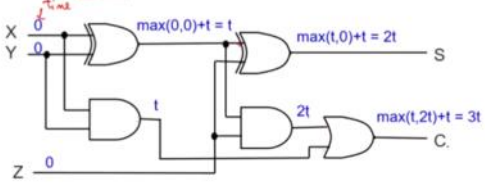
  $\max( t_1, t_2, \ldots, t_n ) + t$




- To calculate the delays of all outputs of a combinational circuit, repeat above rule for all gates.

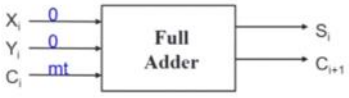
## 9. Circuit Delays (2/5)

- As a simple example, consider the full adder circuit where all inputs are available at time 0. Assume each gate has delay t.



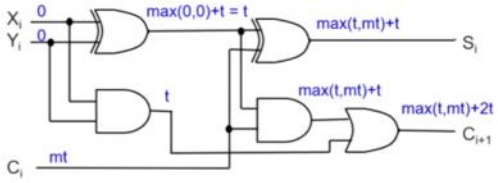

## 9. Circuit Delays (4/5)

- Analyse the delay for the repeated block.

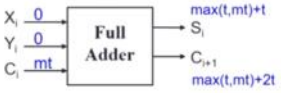



where $X_i$, $Y_i$ are stable at 0t, while $C_i$ is assumed to be stable at mt.

- Performing the delay calculation:




## 9. Circuit Delays (5/5)




- Calculating:
  - When i=1, m=0; $S_1 = 2t$ and $C_2 = 3t$
  - When i=2, m=3; $S_2 = 4t$ and $C_3 = 5t$
  - When i=3, m=5; $S_3 = 6t$ and $C_4 = 7t$
  - When i=4, m=7; $S_4 = 8t$ and $C_5 = 9t$
- In general, an $n$-bit ripple-carry parallel adder will experience the following delay times:
  - $S_n = ( (n - 1)2 + 2 ) t$
  - $C_{n+1} = ( (n - 1)2 + 3 ) t$
- Propagation delay of ripple-carry parallel adders is proportional to the number of bits it handles.

# Week 9

Sunday, October 22, 2023 6:56 PM

## 1. Introduction (1/2)

- An integrated circuit (referred to as an IC, a chip or a microchip) is a set of electronic circuits on one small flat piece (or 'chip') of semiconductor material.
- Scale of integration: the number of components fitted into a standard size IC

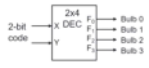
| Name | Signification | Year | #transistors | #logic gates |
|---|---|---|---|---|
| SSI | Small-scale integration | 1964 | 1 to 10 | 1 to 12 |
| MSI | Medium-scale integration | 1968 | 10 to 500 | 13 to 99 |
| LSI | Large-scale integration | 1971 | 500 to 20000 | 100 to 9999 |
| VLSI | Very large-scale integration | 1980 | 20k to 1m | 10k to 99999 |
| ULSI | Ultra-large-scale integration | 1984 | 1m and more | 100k and more |

## 2. Decoders (1/5)

- Codes are frequently used to represent entities, eg: your name is a code to denote yourself (an entity!)
- These codes can be identified (or decoded) using a decoder. Given a code, identify the entity.
- Convert binary information from *n* input lines to (a maximum of) $2^n$ output lines.
- Known as *n*-to-*m*-line decoder, or simply *n*:*m* or *n*×*m* decoder ($m \leq 2^n$).
- May be used to generate $2^n$ minterms of *n* input variables.

## 2. Decoders (2/5)

- Example: If codes 00, 01, 10, 11 are used to identify four light bulbs, we may use a 2-bit decoder.



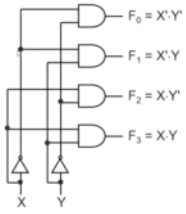

- This is a 2×4 decoder which selects an output line based on the 2-bit code supplied.
- Truth table:

| X | Y | $F_0$ | $F_1$ | $F_2$ | $F_3$ |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |

## 2. Decoders (3/5)

- From truth table, circuit for 2×4 decoder is:
- Note: Each output is a minterm (X'·Y', X'·Y, X·Y' or X·Y) of a 2-variable function

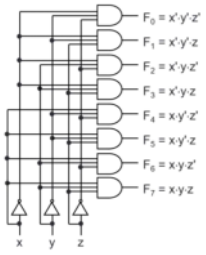
| X | Y | $F_0$ | $F_1$ | $F_2$ | $F_3$ |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |




## 2. Decoders (4/5)

- Design a 3×8 decoder.

| x | y | z | $F_0$ | $F_1$ | $F_2$ | $F_3$ | $F_4$ | $F_5$ | $F_6$ | $F_7$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |



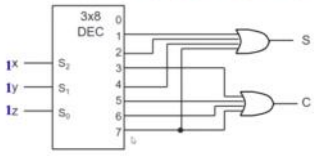

For an n-bit code, a decoder could select up to 2^n lines for output

## 2. Decoders: Implementing Functions (1/3)

- A Boolean function, in sum-of-minterms form ⇨
  - decoder to generate the minterms, and
  - an OR gate to form the sum.
- Any combinational circuit with *n* inputs and *m* outputs can be implemented with an $n$:$2^n$ decoder with *m* OR gates.
- Good when circuit has many outputs, and each function is expressed with a few minterms.

## 2. Decoders: Implementing Functions (3/3)

S(x, y, z) = Σ m(1,2,4,7)
C(x, y, z) = Σ m(3,5,6,7)




| x | y | z | C | S |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

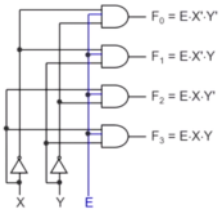
The numbers 0,1,2 … 7 c0me from ordering in truth table
They are the minterm number

## 2. Decoders with Enable (1/2)

- Decoders often come with an *enable* control signal, so that the device is only activated when the enable, E = 1.
- Truth table:

| E | X | Y | $F_0$ | $F_1$ | $F_2$ | $F_3$ |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 0 | d | d | 0 | 0 | 0 | 0 |

- Circuit of a 2×4 decoder with enable:




## 2. Decoders with Enable (2/2)

- In the previous slide, the decoder has a one-enable control signal, i.e. the decoder is enabled with E=1.
- In most MSI decoders, enable signal is zero-enable, usually denoted by E' or Ê. The decoder is enabled when the signal is zero (low).

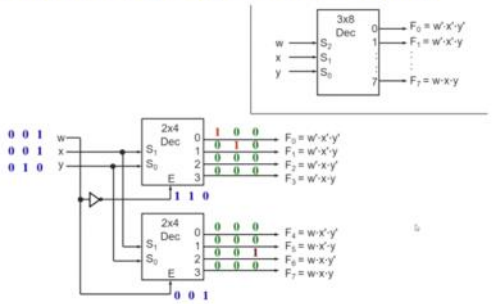
| E | X | Y | $F_0$ | $F_1$ | $F_2$ | $F_3$ |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 0 | d | d | 0 | 0 | 0 | 0 |

Decoder with 1-enable

| E' | X | Y | $F_0$ | $F_1$ | $F_2$ | $F_3$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | d | d | 0 | 0 | 0 | 0 |

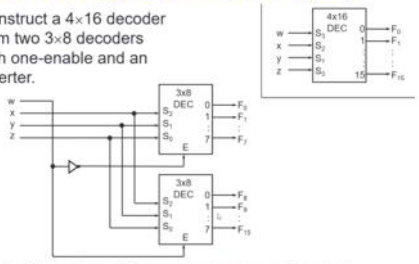
Decoder with 0-enable

## 2. Constructing Larger Decoders (2/4)




## 2. Constructing Larger Decoders (3/4)

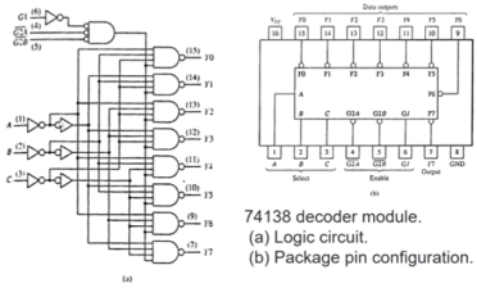
- Construct a 4×16 decoder from two 3×8 decoders with one-enable and an inverter.




- Note: The input w and its complement w' are used to select either one of the two smaller decoders.

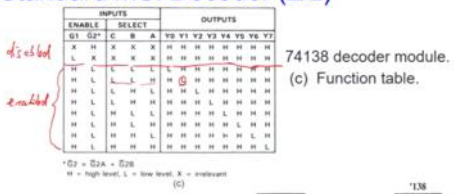
## 2. Standard MSI Decoder (1/2)

- 74138 (3-to-8 decoder)



74138 decoder module.
(a) Logic circuit.
(b) Package pin configuration.

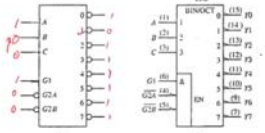
## 2. Standard MSI Decoder (2/2)



74138 decoder module.
(c) Function table.

74138 decoder module.
(d) Generic symbol.
(e) IEEE standard logic symbol.

*Source:The Data Book Volume 2, Texas Instruments Inc., 1985*



For CBA = 001,
We get Y1 = 0 and others = 1 which is opposite of previous ex.
We are looking for the 0

## 2. Decoders: Implementing Functions Revisit (1/2)

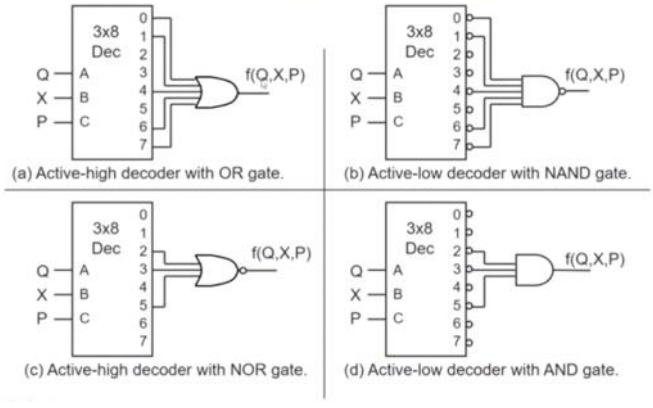
- Example: Implement the following function using a 3×8 decoder and an appropriate logic gate

$$f(Q,X,P) = \sum m(0,1,4,6,7) = \prod M(2,3,5)$$

- We may implement the function in several ways:
  - Using a decoder with active-high outputs with an OR gate:
    $f(Q,X,P) = m_0 + m_1 + m_4 + m_6 + m_7$
  - Using a decoder with active-low outputs with a NAND gate:
    $f(Q,X,P) = (m_0' \cdot m_1' \cdot m_4' \cdot m_6' \cdot m_7')'$
  - Using a decoder with active-high outputs with a NOR gate:
    $f(Q,X,P) = (m_2 + m_3 + m_5)' \; [= M_2 \cdot M_3 \cdot M_5]$
  - Using a decoder with active-low outputs with an AND gate:
    $f(Q,X,P) = m_2' \cdot m_3' \cdot m_5'$

## 2. Decoders: Implementing Functions Revisit (2/2)

$f(Q,X,P) = \Sigma m(0,1,4,6,7) = \prod M(2,3,5)$



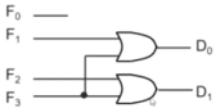

(a) Active-high decoder with OR gate.
(b) Active-low decoder with NAND gate.
(c) Active-high decoder with NOR gate.
(d) Active-low decoder with AND gate.

ENCODERS

Converse of decoding
Given a set of input lines, of which exactly 1 is high and rest are low, encoder provides a code that corresponds to that high input line
Contains 2^n or fewer input lines and n output lines
Implemented iwht OR gates

# 3. Encoders (2/4)

- Truth table:
- With K-map, we obtain:
  - $D_0 = F1 + F3$
    $D_1 = F2 + F3$
- Circuit:




Simple 4-to-2 encoder

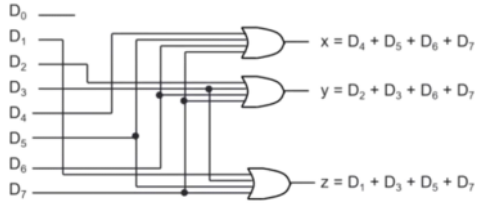
| $F_0$ | $F_1$ | $F_2$ | $F_3$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | X | X |
| 0 | 0 | 1 | 1 | X | X |
| 0 | 1 | 0 | 1 | X | X |
| 0 | 1 | 1 | 0 | X | X |
| 0 | 1 | 1 | 1 | X | X |
| 1 | 0 | 0 | 1 | X | X |
| 1 | 0 | 1 | 0 | X | X |
| 1 | 0 | 1 | 1 | X | X |
| 1 | 1 | 0 | 0 | X | X |
| 1 | 1 | 0 | 1 | X | X |
| 1 | 1 | 1 | 0 | X | X |
| 1 | 1 | 1 | 1 | X | X |

# 3. Encoders (3/4)

- Example: 8-to-3 encoder.
  - At any one time, only one input line of an encoder has a value of 1 (high), the rest are zeroes (low).
  - To allow for more than one input line to carry a 1,we need priority encoder.

| Inputs | | | | | | | | Outputs | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | $D_6$ | $D_7$ | x | y | z |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

- Example: 8-to-3 encoder.




An 8-to-3 encoder

# 3. Priority Encoders (1/2)

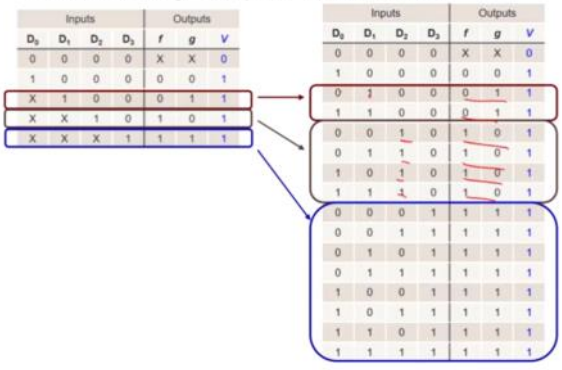
- A priority encoder is one with priority
  - If two or more inputs or equal to 1, the input with the highest priority takes precedence.
  - If all inputs are 0, this input combination is considered invalid.
- Example of a 4-to-2 priority encoder:

| Inputs | | | | Outputs | | |
|---|---|---|---|---|---|---|
| $D_0$ | $D_1$ | $D_2$ | $D_3$ | *f* | *g* | *V* |
| 0 | 0 | 0 | 0 | X | X | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| X | 1 | 0 | 0 | 0 | 1 | 1 |
| X | X | 1 | 0 | 1 | 0 | 1 |
| X | X | X | 1 | 1 | 1 | 1 |

Represents 2 rows with don't care values

# 3. Priority Encoders (2/2)

- Understanding "compact" function table




Multiplexors and demultiplexers

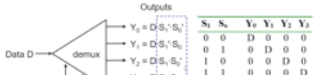
## Multiplexers and Demultiplexers

- An application:

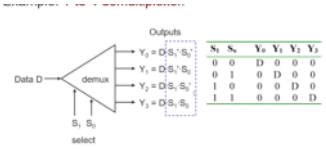
- Helps share a *single communication line* among a number of devices.
- At any time, only one source and one destination can use the communication line.

## 4. Demultiplexers (1/2)

- Given an input line and a set of selection lines, a demultiplexer directs data from the input to one selected output line.
- Example: 1-to-4 demultiplexer.



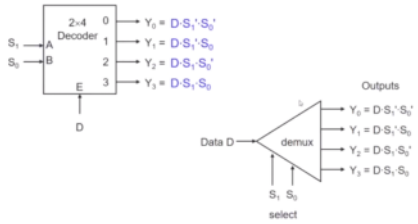
- Helps share a *single communication line* among a number of devices.
- At any time, only one source and one destination can use the communication line.




| $S_1$ | $S_0$ | $Y_0$ | $Y_1$ | $Y_2$ | $Y_3$ |
|---|---|---|---|---|---|
| 0 | 0 | D | 0 | 0 | 0 |
| 0 | 1 | 0 | D | 0 | 0 |
| 1 | 0 | 0 | 0 | D | 0 |
| 1 | 1 | 0 | 0 | 0 | D |

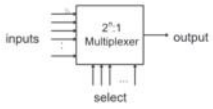
## 4. Demultiplexers (2/2)

- It turns out that the demultiplexer circuit is actually identical to a decoder with enable.




## 5. Multiplexers (1/4)

- A multiplexer is a device that has
  - A number of input lines
  - A number of selection lines
  - One output line
- It steers one of $2^n$ inputs to a single output line, using $n$ selection lines. Also known as a *data selector*.

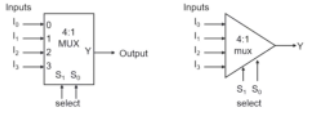



## 5. Multiplexers (2/4)

- Truth table for a 4-to-1 multiplexer:

| $I_0$ | $I_1$ | $I_2$ | $I_3$ | $S_1$ | $S_0$ | Y |
|---|---|---|---|---|---|---|
| $d_0$ | $d_1$ | $d_2$ | $d_3$ | 0 | 0 | $d_0$ |
| $d_0$ | $d_1$ | $d_2$ | $d_3$ | 0 | 1 | $d_1$ |
| $d_0$ | $d_1$ | $d_2$ | $d_3$ | 1 | 0 | $d_2$ |
| $d_0$ | $d_1$ | $d_2$ | $d_3$ | 1 | 1 | $d_3$ |

| $S_1$ | $S_0$ | Y |
|---|---|---|
| 0 | 0 | $I_0$ |
| 0 | 1 | $I_1$ |
| 1 | 0 | $I_2$ |
| 1 | 1 | $I_3$ |




## 5. Multiplexers (3/4)

| $S_1$ | $S_0$ | Y |
|---|---|---|
| 0 | 0 | $I_0$ |
| 0 | 1 | $I_1$ |
| 1 | 0 | $I_2$ |
| 1 | 1 | $I_3$ |

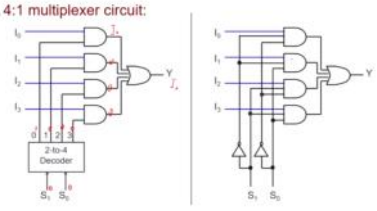
- Output of multiplexer is
  "sum of the (product of *data lines* and *selection lines*)"
- Example: Output of a 4-to-1 multiplexer is:

$$Y = I_0 \cdot (S_1' \cdot S_0') + I_1 \cdot (S_1' \cdot S_0) + I_2 \cdot (S_1 \cdot S_0') + I_3 \cdot (S_1 \cdot S_0)$$

*Note:*
Expressing
$I_0 \cdot (S_1' \cdot S_0') + I_1 \cdot (S_1' \cdot S_0) + I_2 \cdot (S_1 \cdot S_0') + I_3 \cdot (S_1 \cdot S_0)$
in minterms notation, it is equal to
$I_0 \cdot m_0 + I_1 \cdot m_1 + I_2 \cdot m_2 + I_3 \cdot m_3$
This is useful later (eg: slide 45).

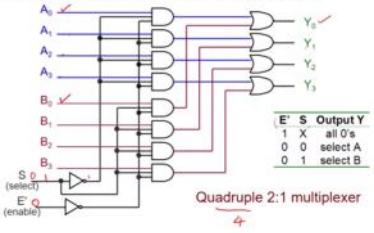
## 5. Multiplexers (4/4)

- A $2^n$-to-1-line multiplexer, or simply $2^n$:1 MUX, is made from an $n$:$2^n$ decoder by adding to it $2^n$ input lines, one to each AND gate.
- A 4:1 multiplexer circuit:



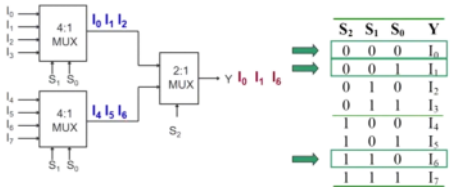

## 5. Multiplexer IC Package

- Some IC packages have a few multiplexers in each package (chip). The selection and enable inputs are common to all multiplexers within the package.




| E' | S | Output Y |
|---|---|---|
| 1 | X | all 0's |
| 0 | 0 | select A |
| 0 | 1 | select B |

## 5. Constructing Larger Multiplexers (2/4)

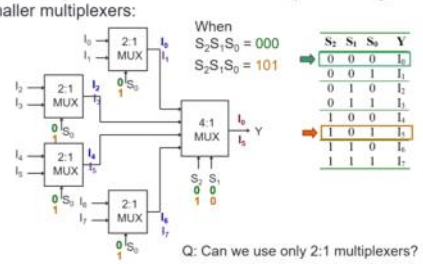



| $S_2$ | $S_1$ | $S_0$ | Y |
|---|---|---|---|
| 0 | 0 | 0 | $I_0$ |
| 0 | 0 | 1 | $I_1$ |
| 0 | 1 | 0 | $I_2$ |
| 0 | 1 | 1 | $I_3$ |
| 1 | 0 | 0 | $I_4$ |
| 1 | 0 | 1 | $I_5$ |
| 1 | 1 | 0 | $I_6$ |
| 1 | 1 | 1 | $I_7$ |

- When $S_2S_1S_0 = 000$
- When $S_2S_1S_0 = 001$
- When $S_2S_1S_0 = 110$

## 5. Constructing Larger Multiplexers (3/4)

- Another implementation of an 8-to-1 multiplexer using smaller multiplexers:

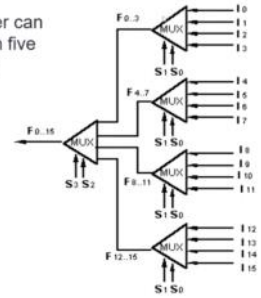



| $S_2$ | $S_1$ | $S_0$ | Y |
|---|---|---|---|
| 0 | 0 | 0 | $I_0$ |
| 0 | 0 | 1 | $I_1$ |
| 0 | 1 | 0 | $I_2$ |
| 0 | 1 | 1 | $I_3$ |
| 1 | 0 | 0 | $I_4$ |
| 1 | 0 | 1 | $I_5$ |
| 1 | 1 | 0 | $I_6$ |
| 1 | 1 | 1 | $I_7$ |

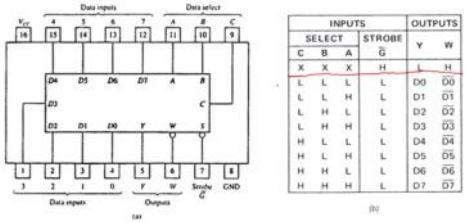
yes

## 5. Constructing Larger Multiplexers (4/4)

- A 16-to-1 multiplexer can be constructed from five 4-to-1 multiplexers:




## 5. Standard MSI Multiplexer (1/2)



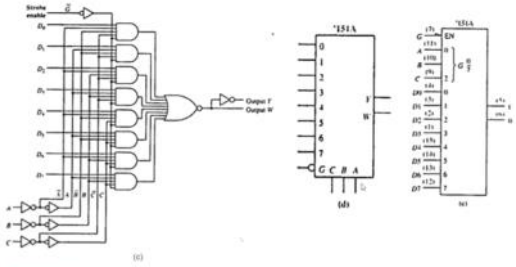
| INPUTS | | | | OUTPUTS | |
|---|---|---|---|---|---|
| SELECT | | | STROBE | | |
| C | B | A | $\overline{G}$ | Y | W |
| X | X | X | H | L | H |
| L | L | L | L | D0 | $\overline{D0}$ |
| L | L | H | L | D1 | $\overline{D1}$ |
| L | H | L | L | D2 | $\overline{D2}$ |
| L | H | H | L | D3 | $\overline{D3}$ |
| H | L | L | L | D4 | $\overline{D4}$ |
| H | L | H | L | D5 | $\overline{D5}$ |
| H | H | L | L | D6 | $\overline{D6}$ |
| H | H | H | L | D7 | $\overline{D7}$ |

**74151A 8-to-1 multiplexer. (a) Package configuration. (b) Function table.**

Strobe is enabler. Its a 0 enabler. 0 for strobe will enable

C is msb

## 5. Standard MSI Multiplexer (2/2)



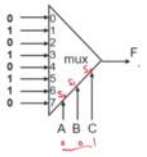
**74151A 8-to-1 multiplexer**. (c) Logic diagram. (d) Generic logic symbol. (e) IEEE standard logic symbol.
*Source: The TTL Data Book Volume 2. Texas Instruments Inc.,1985.*

## 5. Multiplexers: Implementing Functions (1/3)

- Boolean functions can be implemented using multiplexers.
- A $2^n$-to-1 multiplexer can implement a Boolean function of *n* input variables, as follows:
  1. Express in sum-of-minterms form.
     Example:
     $F(A,B,C) = A'\cdot B'\cdot C + A'\cdot B\cdot C + A\cdot B'\cdot C + A\cdot B\cdot C'$
     $= \Sigma\, m(1,3,5,6)$
  2. Connect *n* variables to the *n* selection lines.
  3. Put a '1' on a data line if it is a minterm of the function, or '0' otherwise.

## 5. Multiplexers: Implementing Functions (2/3)

- $F(A,B,C) = \Sigma\, m(1,3,5,6)$




This method works because:

Output = $I_0\cdot m_0 + I_1\cdot m_1 + I_2\cdot m_2 + I_3\cdot m_3 + I_4\cdot m_4 + I_5\cdot m_5 + I_6\cdot m_6 + I_7\cdot m_7$

Supplying '1' to $I_1, I_3, I_5, I_6$ , and '0' to the rest:

Output = $m_1 + m_3 + m_5 + m_6$

ABC corresponds to a minterm

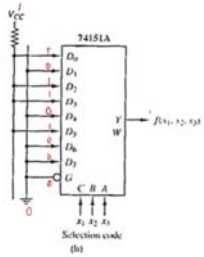
*From slide 34 (4:1 mux)*
Expressing
$I_0\cdot(S_1'\cdot S_0') + I_1\cdot(S_1'\cdot S_0) + I_2\cdot(S_1\cdot S_0') + I_3\cdot(S_1\cdot S_0)$
in minterms notation, it is equal to
$I_0\cdot m_0 + I_1\cdot m_1 + I_2\cdot m_2 + I_3\cdot m_3$

## 5. Multiplexers: Implementing Functions (3/3)

- Example: Use a 74151A to implement
  $f(x_1,x_2,x_3) = \Sigma\, m(0,2,3,5)$

| i | C $x_1$ | B $x_2$ | A $x_3$ | Y f |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 2 | 0 | 1 | 0 | 1 |
| 3 | 0 | 1 | 1 | 1 |
| 4 | 1 | 0 | 0 | 0 |
| 5 | 1 | 0 | 1 | 1 |
| 6 | 1 | 1 | 0 | 0 |
| 7 | 1 | 1 | 1 | 0 |

(a)



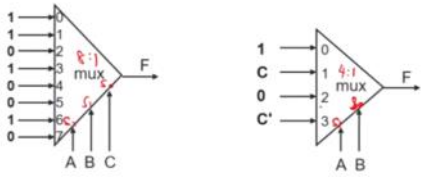

Realization of $f(x_1,x_2,x_3) = \Sigma m(0,2,3,5)$.
(a) Truth table.
(b) Implementation with 74151A.

## 5. Using Smaller Multiplexers (1/6)

- Earlier, we saw how a $2^n$-to-1 multiplexer can be used to implement a Boolean function of *n* (input) variables.
- However, we can use a single smaller $2^{(n-1)}$-to-1 multiplexer to implement a Boolean function of *n* (input) variables.
- Example: The function
  $F(A,B,C) = \Sigma\, m(1,3,5,6)$
  can be implemented using a 4-to-1 multiplexer (rather than an 8-to-1 multiplexer).

## 5. Using Smaller Multiplexers (2/6)

- Let's look at this example:
  $F(A,B,C) = \Sigma\, m(0,1,3,6) = A'\cdot B'\cdot C' + A'\cdot B'\cdot C + A'\cdot B\cdot C + A\cdot B\cdot C'$




- Note: Two of the variables, A and B, are applied as selection lines of the multiplexer, while the inputs of the multiplexer contain 1, C, 0 and C'.

4 to 1 multiplexer is the next smallest after 8 to 1

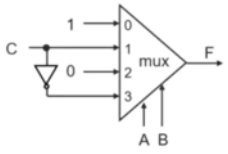
## 5. Using Smaller Multiplexers (3/6)

- Procedure
  1. Express Boolean function in sum-of-minterms form.
     Example: $F(A,B,C) = \Sigma\, m(0,1,3,6)$
  2. Reserve one variable (in our example, we take the least significant one) for input lines of multiplexer, and use the rest for selection lines.
     Example: C is for input lines; A and B for selection lines.

## 5. Using Smaller Multiplexers (4/6)

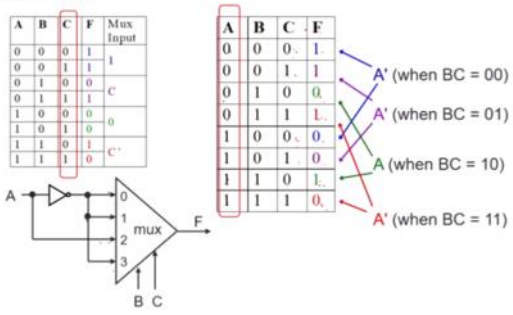
3. Draw the truth table for function, by grouping inputs by selection line values, then determine multiplexer inputs by comparing input line (C) and function (F) for corresponding selection line values.

| 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 | |



## 5. Using Smaller Multiplexers (5/6)

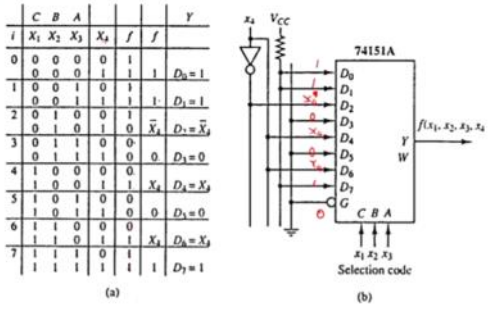
- Alternative: What if we use A for input lines, and B, C for selector lines?




## 5. Using Smaller Multiplexers (6/6)

- Example: Implement the function below with 74151A:

$f(x_1,x_2,x_3,x_4) = \Sigma\, m(0,1,2,3,4,9,13,14,15)$



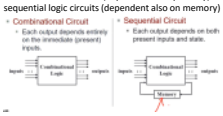
# Week 10

Tuesday, October 31, 2023 12:51 AM

Sequential logic
Latches and Flip flops

Logic Circuits

- There are combinations (dependent on input only) and sequential logic circuits (dependent also on memory)




Multivibrator: a class of sequential circuits

- There are bistable (2 states), monostable, and astable

Bistable logic devices - latches and flip flops

## 2. Memory Elements (1/3)

- Memory element: a device which can remember value indefinitely, or change value on command from its inputs.

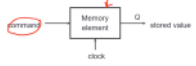

- Characteristic table:

| Command (at time $t_i$) | $Q(t)$ or $Q_i$ | $Q(t+1)$ or $Q_{i+1}$ |
|---|---|---|
| Set | X | 1 |
| Reset | X | 0 |
| Memorize / No Change | 0 | 0 |
| | 1 | 1 |

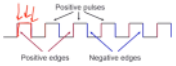
$Q(t)$ or $Q_i$: current state
$Q(t+1)$ or $Q_{i+1}$: next state

## 2. Memory Elements (2/3)

- Memory element with clock.




- Clock is usually a square wave.




PC is edge triggered

## 2. Memory Elements (3/3)

- Two types of triggering/activation
  - Pulse-triggered
  - Edge-triggered
- Pulse-triggered
  - Latches
  - ON = 1, OFF = 0


- Edge-triggered
  - Flip-flops
  - Positive edge-triggered (ON = from 0 to 1; OFF = other time)
  - Negative edge-triggered (ON = from 1 to 0; OFF = other time)

Latches are only active when high state

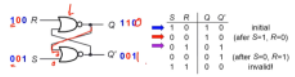
LATCHES

## 3.1 S-R Latch (1/3)

- Two inputs: $S$ and $R$.
- Two complementary outputs: $Q$ and $Q'$.
  - When $Q$ = HIGH, we say latch is in SET state.
  - When $Q$ = LOW, we say latch is in RESET state.
- For active-high input $S$-$R$ latch (also known as NOR gate latch)
  - $R$ = HIGH and $S$ = LOW → $Q$ becomes LOW (RESET state)
  - $S$ = HIGH and $R$ = LOW → $Q$ becomes HIGH (SET state)
  - Both $R$ and $S$ are LOW →No change in output $Q$
  - Both $R$ and $S$ are HIGH →Outputs $Q$ and $Q'$ are both LOW (invalid!)
- Drawback: invalid condition exists and must be avoided.

## 3.1 S-R Latch (2/3)

- Active-high input $S$-$R$ latch:




| $S$ | $R$ | $Q$ | $Q'$ | |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | initial |
| 0 | 0 | 1 | 0 | (after $S$=1, $R$=0) |
| 0 | 1 | 0 | 1 | |
| 0 | 0 | 0 | 1 | (after $S$=0, $R$=1) |
| 1 | 1 | 0 | 0 | invalid |

## 3.1 S-R Latch (3/3)

- Characteristic table for active-high input $S$-$R$ latch:


| $S$ | $R$ | $Q$ | $Q'$ | |
|---|---|---|---|---|
| 0 | 0 | NC | NC | No change. Latch remained in present state. |
| 1 | 0 | 1 | 0 | Latch SET. |
| 0 | 1 | 0 | 1 | Latch RESET. |
| 1 | 1 | 0 | 0 | Invalid condition. |

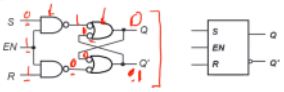
| $S$ | $R$ | $Q(t+1)$ | |
|---|---|---|---|
| 0 | 0 | $Q(t)$ | No change |
| 0 | 1 | 0 | Reset |
| 1 | 0 | 1 | Set |
| 1 | 1 | indeterminate | |

$Q(t+1) = S + R'Q$
$S \cdot R = 0$

- Block diagram:


## 3.1 Gated S-R Latch

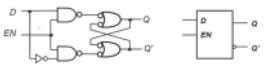
- $S$-$R$ latch + *enable input* ($EN$) and 2 NAND gates → a gated $S$-$R$ latch.




- Outputs change (if necessary) only when $EN$ is high.

## 3.2 Gated D Latch (1/2)

- Make input $R$ equal to $S'$ → gated $D$ latch.
- $D$ latch eliminates the undesirable condition of invalid state in the $S$-$R$ latch.




## 3.2 Gated D Latch (2/2)

- When $EN$ is high,
  - $D$ = HIGH → latch is SET
  - $D$ = LOW → latch is RESET
- Hence when EN is high, $Q$ "follows" the $D$ (data) input.
- Characteristic table:

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

When $EN$=1, $Q(t+1) = ?$

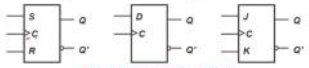
FLIP FLOPS

## 4. Flip-flops (1/2)

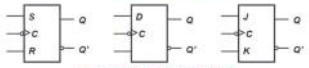
- Flip-flops are synchronous bistable devices.
- Output changes state at a specified point on a triggering input called the clock.
- Change state either at the positive (rising) edge, or at the negative (falling) edge of the clock signal.


## 4. Flip-flops (2/2)

- *S-R* flip-flop, *D* flip-flop, and *J-K* flip-flop.
- Note the ">" symbol at the clock input.




Positive edge-triggered flip-flops




Negative edge-triggered flip-flops

## 4.1 *S-R* Flip-flop

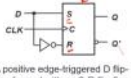
- *S-R* flip-flop: On the triggering edge of the clock pulse,
  - *R* = HIGH and *S* = LOW → *Q* becomes LOW (RESET state)
  - *S* = HIGH and *R* = LOW → *Q* becomes HIGH (SET state)
  - Both *R* and *S* are LOW →No change in output *Q*
  - Both *R* and *S* are HIGH →Invalid!
- Characteristic table of positive edge-triggered *S-R* flip-flop:


| S | R | CLK | Q(t+1) | Comments |
|---|---|---|---|---|
| 0 | 0 | X | Q(t) | No change |
| 0 | 1 | ↑ | 0 | Reset |
| 1 | 0 | ↑ | 1 | Set |
| 1 | 1 | ↑ | ? | Invalid |

X = irrelevant ("don't care")
↑ = clock transition LOW to HIGH

## 4.2 *D* Flip-flop (1/2)

- *D* flip-flop: Single input *D* (data). On the triggering edge of the clock pulse,
  - *D* = HIGH → *Q* becomes HIGH (SET state)
  - *D* = LOW → *Q* becomes LOW (RESET state)
- Hence, *Q* "follows" *D* at the clock edge.
- Convert *S-R* flip-flop into a *D* flip-flop: add an inverter.



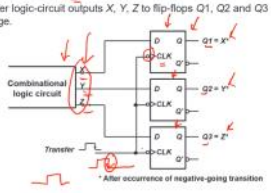

A positive edge-triggered D flip-flop formed with an S-R flip-flop.

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

↑ = clock transition LOW to HIGH

## 4.2 *D* Flip-flop (2/2)

- Application: Parallel data transfer.
  - To transfer logic-circuit outputs *X*, *Y*, *Z* to flip-flops *Q*1, *Q*2 and *Q*3 for storage.



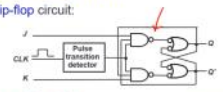

Will only change value at pulse

## 4.3 *J-K* Flip-flop (1/2)

- *J-K* flip-flop: *Q* and *Q'* are fed back to the pulse-steering NAND gates.
- No invalid state.
- Include a toggle state
  - *J* = HIGH and *K* = LOW → *Q* becomes HIGH (SET state)
  - *K* = HIGH and *J* = LOW → *Q* becomes LOW (RESET state)
  - Both *J* and *K* are LOW →No change in output *Q*
  - Both *J* and *K* are HIGH →Toggle

## 4.3 *J-K* Flip-flop (2/2)

- *J-K* flip-flop circuit:




- Characteristic table:

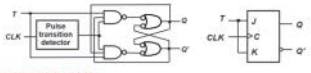
| J | K | CLK | Q(t+1) | Comments |
|---|---|---|---|---|
| 0 | 0 | ↑ | Q(t) | No change |
| 0 | 1 | ↑ | 0 | Reset |
| 1 | 0 | ↑ | 1 | Set |
| 1 | 1 | ↑ | Q(t)' | Toggle |

Q(t+1) = J.Q' + K'.Q

| Q | J | K | Q(t+1) |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

## 4.4 *T* Flip-flop

- *T* flip-flop: Single input version of the *J-K* flip-flop, formed by tying both inputs together.




- Characteristic table:

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

Q(t+1) = ?

| Q | T | Q(t+1) |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

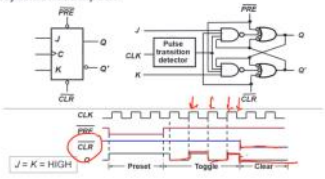
T flip flops can only toggle or not
D filip flops can change or reset

## 5. Asynchronous Inputs (1/2)

- *S-R*, *D* and *J-K* inputs are synchronous inputs, as data on these inputs are transferred to the flip-flop's output only on the triggered edge of the clock pulse.
- Asynchronous inputs affect the state of the flip-flop independent of the clock; example: *preset* (*PRE*) and *clear* (*CLR*) [or *direct set* (*SD*) and *direct reset* (*RD*)].
- When *PRE*=HIGH, *Q* is immediately set to HIGH.
- When *CLR*=HIGH, *Q* is immediately cleared to LOW.
- Flip-flop in normal operation mode when both *PRE* and *CLR* are LOW.

## 5. Asynchronous Inputs (2/2)

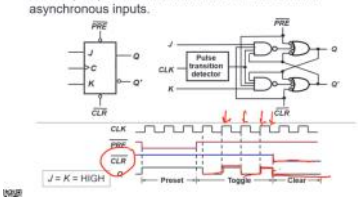
- A *J-K* flip-flop with active-low PRESET and CLEAR asynchronous inputs.



## 5. Asynchronous Inputs (1/2)

- *S-R*, *D* and *J-K* inputs are synchronous inputs, as data on these inputs are transferred to the flip-flop's output only on the triggered edge of the clock pulse.
- Asynchronous inputs affect the state of the flip-flop independent of the clock; example: *preset* (*PRE*) and *clear* (*CLR*) [or *direct set* (*SD*) and *direct reset* (*RD*)].
- When *PRE*=HIGH, *Q* is immediately set to HIGH.
- When *CLR*=HIGH, *Q* is immediately cleared to LOW.
- Flip-flop in normal operation mode when both *PRE* and *CLR* are LOW.

- A *J-K* flip-flop with active-low PRESET and CLEAR asynchronous inputs.




## 6.2 Sequential Circuits: Analysis (1/7)

- Given a sequential circuit diagram, we can analyze its behaviour by deriving its *state table* and hence its *state diagram*.
- Requires *state equations* to be derived for the flip-flop inputs, as well as *output functions* for the circuit outputs other than the flip-flops (if any).
- We use ***A(t)*** and ***A(t+1)*** (or simply ***A*** and ***A*⁺**) to represent the present state and next state, respectively, of a flip-flop represented by *A*.

## 6.2 Sequential Circuits: Analysis (2/7)

- Example using *D* flip-flops

State equations:
$A^+ = A \cdot x + B \cdot x$
$B^+ = A' \cdot x$
Output function:
$y = (A + B) \cdot x'$

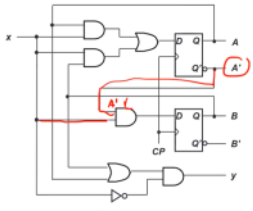



Figure 1

## 6.2 Sequential Circuits: Analysis (3/7)

- From the *state equations* and *output function*, we derive the ***state table***, consisting of all possible binary combinations of present states and inputs.
- State table
  - Similar to truth table.
  - Inputs and present state on the left side.
  - Outputs and next state on the right side.
- *m* flip-flops and *n* inputs → $2^{m+n}$ rows.

## 6.2 Sequential Circuits: Analysis (4/7)

- ***State table*** for circuit of Figure 1:

State equations:
$A^+ = A \cdot x + B \cdot x$
$B^+ = A' \cdot x$

Output function:
$y = (A + B) \cdot x'$

| Present State A | Present State B | Input x | Next State $A^+$ | Next State $B^+$ | Output y |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 |

## 6.2 Sequential Circuits: Analysis (5/7)

- Alternative form of state table:

Full table

| Present State A | Present State B | Input x | Next State $A^+$ | Next State $B^+$ | Output y |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 |

Compact table

| Present State AB | Next State x=0 $A^+B^+$ | Next State x=1 $A^+B^+$ | Output x=0 y | Output x=1 y |
|---|---|---|---|---|
| 00 | 00 | 01 | 0 | 0 |
| 01 | 00 | 11 | 1 | 0 |
| 10 | 00 | 10 | 1 | 0 |
| 11 | 00 | 10 | 1 | 0 |

## 6.2 Sequential Circuits: Analysis (6/7)

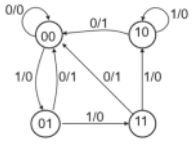
- From the *state table*, we can draw the ***state diagram***.
- State diagram
  - Each state is denoted by a circle.
  - Each arrow (between two circles) denotes a transition of the sequential circuit (a row in state table).
  - A label of the form *a*/*b* is attached to each arrow where *a* (if there is one) denotes the inputs while *b* (if there is one) denotes the outputs of the circuit in that transition.
- Each combination of the flip-flop values represents a state. Hence, *m* flip-flops → up to $2^m$ states.

## 6.2 Sequential Circuits: Analysis (7/7)

- **State diagram** of the circuit of Figure 1:

| Present State AB | Next State x=0 $A^+B^+$ | Next State x=1 $A^+B^+$ | Output x=0 y | Output x=1 y |
|---|---|---|---|---|
| 00 | 00 | 01 | 0 | 0 |
| 01 | 00 | 11 | 1 | 0 |
| 10 | 00 | 10 | 1 | 0 |
| 11 | 00 | 10 | 1 | 0 |

DONE!



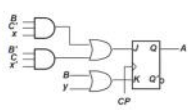

## 6.2 Flip-flop Input Functions (1/3)

- The outputs of a sequential circuit are functions of the present states of the flip-flops and the inputs. These are described algebraically by the *circuit output functions*.
  - In Figure 1: $y = (A + B) \cdot x'$
- The part of the circuit that generates inputs to the flip-flops are described algebraically by the *flip-flop input functions* (or *flip-flop input equations*).
- The flip-flop input functions determine the next state generation.
- From the flip-flop input functions and the characteristic tables of the flip-flops, we obtain the next states of the flip-flops.

## 6.2 Flip-flop Input Functions (2/3)

- Example: circuit with a *JK* flip-flop.
- We use 2 letters to denote each flip-flop input: the first letter denotes the input of the flip-flop (*J* or *K* for *J-K* flip-flop, *S* or *R* for *S-R* flip-flop, *D* for *D* flip-flop, *T* for *T* flip-flop) and the second letter denotes the name of the flip-flop.

$JA = B \cdot C' \cdot x + B' \cdot C \cdot x'$
$KA = B + y$




## 6.2 Analysis: Example #2 (1/3)

- Given Figure 2, a sequential circuit with two *J-K* flip-flops *A* and *B*, and one input *x*.

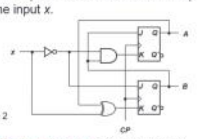



Figure 2

- Obtain the flip-flop input functions from the circuit:

$JA = B$ $\quad JB = x'$
$KA = B \cdot x'$ $\quad KB = A \cdot x' + A \cdot x' = A \oplus x$

## 6.2 Analysis: Example #2 (2/3)

$JA = B$ $\quad JB = x'$
$KA = B \cdot x'$ $\quad KB = A \cdot x' + A \cdot x' = A \oplus x$

- Fill the state table using the above functions, knowing the characteristics of the flip-flops used.

| 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 |

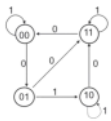
| Present state A | Present state B | Input x | Next state A+ | Next state B+ | Flip-flop inputs JA | Flip-flop inputs KA | Flip-flop inputs JB | Flip-flop inputs KB |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1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

For j/k flip flop, must use table to figure out what is next state

## 6.2 Analysis: Example #2 (3/3)

- Draw the state diagram from the state table.

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




## 6.2 Analysis: Example #3 (1/3)

- Derive the state table and state diagram of this circuit.

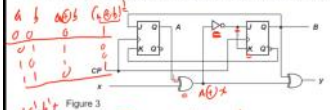



Figure 3

$JA = B$ $\quad KA = B'$

$JB = (A \oplus x)' = A \cdot x + A' \cdot x'$

$KB = JB = (A \oplus x) = A \cdot x' + A' \cdot x$

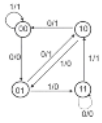
## 6.2 Analysis: Example #3 (2/3)

- Flip-flop input functions:

$JA = B$ $\quad JB = KB = (A \oplus x)' = A \cdot x + A' \cdot x'$
$KA = B'$

- State table:

| Present state A | Present state B | Input x | Next state A+ | Next state B+ | Output y | Flip-flop inputs JA | Flip-flop inputs KA | Flip-flop inputs JB | Flip-flop inputs 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 |




## 6.2 Analysis: Example #3 (1/3)

- Derive the state table and state diagram of this circuit.

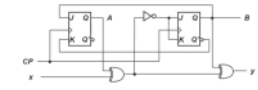



Figure 3

- Flip-flop input functions:

$JA = B$ $\quad JB = KB = (A \oplus x)' = A \cdot x + A' \cdot x'$
$KA = B'$

## 6.3 Flip-flop Excitation Tables (1/2)

- *Excitation tables*: given the required transition from present state to next state, determine the flip-flop input(s).

| 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 |

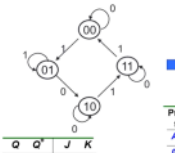
*T* Flip-flop

## 6.4 Sequential Circuits: Design

- Design procedure:
  - Start with circuit specifications – description of circuit behaviour, usually a state diagram or state table.
  - Derive the state table.
  - Perform state reduction if necessary.
  - Perform state assignment.
  - Determine number of flip-flops and label them.
  - Choose the type of flip-flop to be used.
  - Derive circuit excitation and output tables from the state table.
  - Derive circuit output functions and flip-flop input functions.
  - Draw the logic diagram.

## 6.4 Design: Example #1 (2/5)

- Circuit state/excitation table, using *JK* flip-flops.



| Present State | Next State $x=0$ | Next State $x=1$ |
|---|---|---|
| $AB$ | $A^+B^+$ | $A^+B^+$ |
| 00 | 00 | 01 |
| 01 | 10 | 01 |
| 10 | 10 | 11 |
| 11 | 11 | 00 |

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

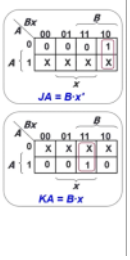
JK Flip-flop's excitation table.

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

## 6.4 Design: Example #1 (4/5)

- From state table, get flip-flop input functions.

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

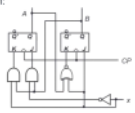


$JA = B \cdot x'$ $KA = B \cdot x$ $JB = x$ $KB = (A \oplus x)'$

A'B' + AB = (A xor B)'

## 6.4 Design: Example #1 (5/5)

- Flip-flop input functions:
  - $JA = B \cdot x'$ $JB = x$
  - $KA = B \cdot x$ $KB = (A \oplus x)'$
- Logic diagram:



## 6.4 Design: Example #2 (1/3)

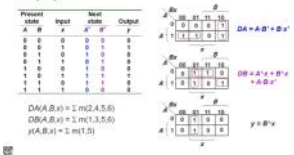
- Using *D* flip-flops, design the circuit based on the state table below. (Exercise: Design it using *JK* flip-flops.)

| Present state $A$ | $B$ | Input $x$ | Next state $A^+$ | $B^+$ | Output $y$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |

Since its D flip flop,

## 6.4 Design: Example #2 (2/3)

- Determine expressions for flip-flop inputs and the circuit output *y*.



$DA(A,B,x) = \Sigma\, m(2,4,5,6)$

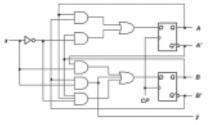
$DB(A,B,x) = \Sigma\, m(1,3,5,6)$

$y(A,B,x) = \Sigma\, m(1,5)$

$DA = A \cdot B' + B \cdot x'$ $DB = A' \cdot x + B' \cdot x + A \cdot B \cdot x'$ $y = B' \cdot x$

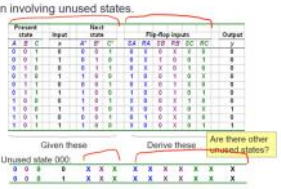
## 6.4 Design: Example #2 (3/3)

- From derived expressions, draw logic diagram:



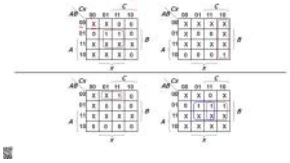
## 6.4 Design: Example #3 (1/4)

- Design involving unused states.



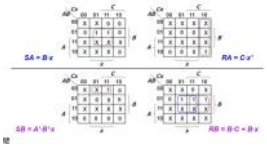

## 6.4 Design: Example #3 (2/4)

- From state table, obtain expressions for flip-flop inputs.



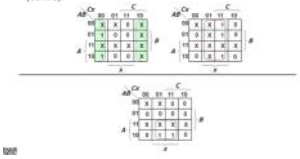
## 6.4 Design: Example #3 (2/4)

- From state table, obtain expressions for flip-flop inputs.




## 6.4 Design: Example #3 (3/4)

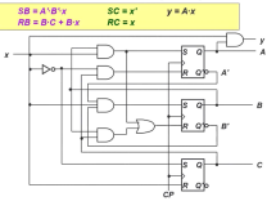
- From state table, obtain expressions for flip-flop inputs (cont'd).



## 6.4 Design: Example #3 (4/4)

- From derived expressions, draw the logic diagram:

$SA = B \cdot x$ $SB = A'B'x$ $SC = x'$ $y = A \cdot x$
$RA = C \cdot x'$ $RB = B \cdot C + B \cdot x$ $RC = x$

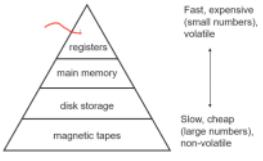


## 7. Memory (1/4)

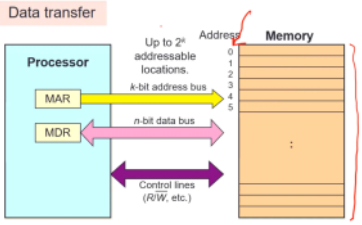
- Memory stores programs and data.
- Definitions:
  - 1 byte = 8 bits
  - 1 word: in multiple of bytes, a unit of transfer between main memory and registers, usually size of register.
  - 1 KB (kilo-bytes) = $2^{10}$ bytes; 1 MB (mega-bytes) = $2^{20}$ bytes; 1 GB (giga-bytes) = $2^{30}$ bytes; 1 TB (tera-bytes) = $2^{40}$ bytes.
- Desirable properties: fast access, large capacity, economical cost, non-volatile.
- However, most memory devices do not possess all these properties.

## 7. Memory (2/4)

Memory hierarchy



## 7. Memory (3/4)




Memory Address Register
Memory Data Register
1 for read/ 0 for write

## 7. Memory (4/4)

- A memory unit stores binary information in groups of bits called *words*.
- The data consists of *n* lines (for *n*-bit words). Data input lines provide the information to be stored (*written*) into the memory, while data output lines carry the information out (*read*) from the memory.
- The address consists of *k* lines which specify which word (among the $2^k$ words available) to be selected for reading or writing.
- The control lines *Read* and *Write* (usually combined into a single control line *Read/Write*) specifies the direction of transfer of the data.

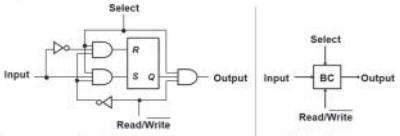
## 7.2 Read/Write Operations

- Write operation:
  - Transfers the address of the desired word to the address lines.
  - Transfers the data bits (the word) to be stored in memory to the data input lines.
  - Activates the *Write* control line (set *Read/Write* to 0).
- Read operation:
  - Transfers the address of the desired word to the address lines.
  - Activates the *Read* control line (set *Read/Write* to 1).

| Memory Enable | *Read/Write* | Memory Operation |
|---|---|---|
| 0 | X | None |
| 1 | 0 | Write to selected word |
| 1 | 1 | Read from selected word |

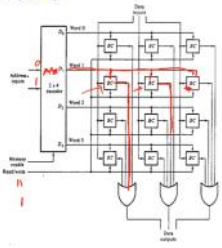
## 7.3 Memory Cell

- Two types of RAM
  - Static RAMs use flip-flops as the memory cells.
  - Dynamic RAMs use capacitor charges to represent data. Though simpler in circuitry, they have to be constantly refreshed.
- A single memory cell of the static RAM has the following logic and block diagrams:



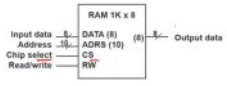

## 7.4 Memory Arrays (1/4)

- Logic construction of a 4×3 RAM (with decoder and OR gates):



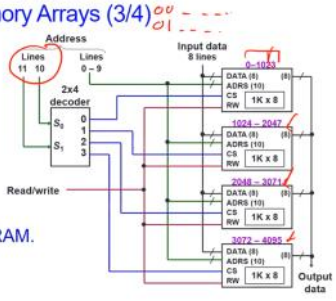
## 7.4 Memory Arrays (2/4)

- An array of RAM chips: memory chips are combined to form larger memory.
- A 1K × 8-bit RAM chip:




Block diagram of a 1K x 8 RAM chip

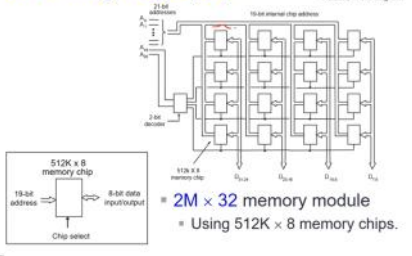
## 7.4 Memory Arrays (3/4)




- 4K × 8 RAM.

First 2 bit determine which ram to activate

## 7.4 Memory Arrays (4/4)




- 2M × 32 memory module
  - Using 512K × 8 memory chips.

32/8 = 4 4 rows needed
2MB/ 512 K = 3.9 smth, 4 columns needed

# Week 11

Monday, November 6, 2023 9:35 PM

Pipelining

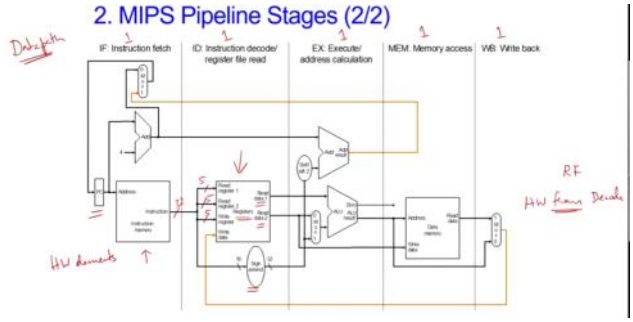
## 1. Introduction: Pipelining Lessons

- Pipelining doesn't help **latency** of single task:
  - It helps the **throughput** of entire workload
- **Multiple** tasks operating simultaneously using different resources
- Possible delays:
  - Pipeline rate limited by **slowest** pipeline stage
  - Stall for **dependencies**

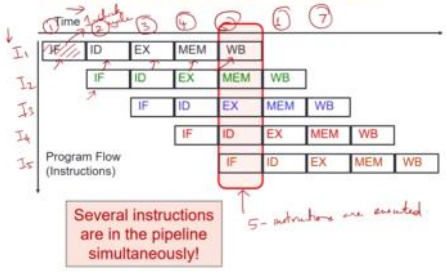
## 2. MIPS Pipeline Stages (1/2)

- **Five** Execution Stages
  - **IF**: Instruction Fetch
  - **ID**: Instruction Decode and Register Read
  - **EX**: Execute an operation or calculate an address
  - **MEM**: Access an operand in data memory
  - **WB**: Write back the result into a register
- **Idea:**
  - **Each execution stage takes 1 clock cycle**
  - General flow of data is from one stage to the next
- **Exceptions:**
  - Update of PC and write back of register file – more about this later…

## 2. MIPS Pipeline Stages (2/2)




## 2. Pipelined Execution: Illustration




It took 5 clock cycles for first instruction to complete, thereafter 1 instruction completes at each subsequent clock cycle

## 3. MIPS Pipeline: Datapath (1/3)

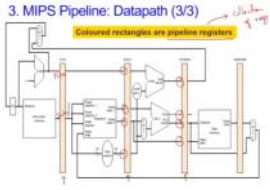
- Single-cycle implementation:
  - Update all state elements (PC, register file, data memory) at the end of a clock cycle
- Pipelined implementation:
  - One cycle per pipeline stage
  - Data required for each stage needs to be stored separately (why?)

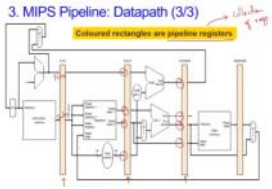
## 3. MIPS Pipeline: Datapath (2/3)

- Data used by **subsequent instructions:**
  - Store in programmer-visible state elements: PC, register file and memory
- Data used by **same instruction** in later pipeline stages:
  - Additional registers in datapath called **pipeline registers**
  - **IF/ID**: register between IF and ID
  - **ID/EX**: register between ID and EX
  - **EX/MEM**: register between EX and MEM
  - **MEM/WB**: register between MEM and WB
- Why no register at the end of **WB** stage?

## 3. MIPS Pipeline: Datapath (3/3)

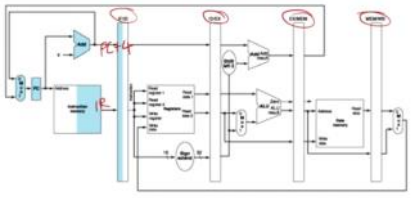
**Coloured rectangles are pipeline registers**





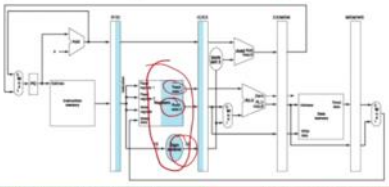

Pipeline datapath & registers

## 3. Pipeline Datapath: IF Stage



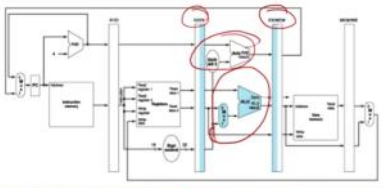
- At the end of a cycle, IF/ID receives (stores):
  - Instruction read from InstructionMemory[ PC ]
  - PC + 4
- PC + 4
  - Also connected to one of the MUX's inputs (another coming later)

## 3. Pipeline Datapath: ID Stage



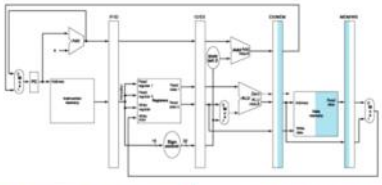
| At the beginning of a cycle IF/ID register supplies: | At the end of a cycle ID/EX receives: |
|---|---|
| ❖ Register numbers for reading two registers<br>❖ 16-bit offset to be sign-extended to 32-bit<br>❖ PC + 4 | ❖ Data values read from register file<br>❖ 32-bit immediate value<br>❖ PC + 4 |

## 3. Pipeline Datapath: EX Stage



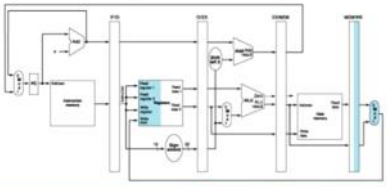
| At the beginning of a cycle ID/EX register supplies: | At the end of a cycle EX/MEM receives: |
|---|---|
| ❖ Data values read from register file<br>❖ 32-bit immediate value<br>❖ PC + 4 | ❖ (PC + 4) + (Immediate x 4)<br>❖ ALU result<br>❖ isZero? signal<br>❖ Data Read 2 from register file |

## 3. Pipeline Datapath: MEM Stage



| At the beginning of a cycle EX/MEM register supplies: | At the end of a cycle MEM/WB receives: |
|---|---|
| ❖ (PC + 4) + (Immediate x 4)<br>❖ ALU result<br>❖ isZero? signal<br>❖ Data Read 2 from register file | ❖ ALU result<br>❖ Memory read data |

## 3. Pipeline Datapath: WB Stage



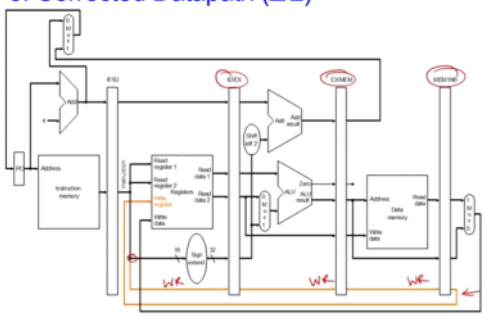
| At the beginning of a cycle MEM/WB register supplies: | At the end of a cycle |
|---|---|
| ❖ ALU result<br>❖ Memory read data | ❖ Result is written back to register file (if applicable)<br>❖ There is a bug here..... |

Write back is not passed in the timeline

## 3. Corrected Datapath (1/2)

- Observe the "Write register" number
  - Supplied by the IF/ID pipeline register
  - → It is NOT the correct write register for the instruction now in WB stage!
- **Solution:**
  - Pass "Write register" number from ID/EX through EX/MEM to MEM/WB pipeline register for use in WB stage
  - i.e. let the "Write register" number follows the instruction through the pipeline until it is needed in WB stage

## 3. Corrected Datapath (2/2)



Control path changes
Each control signal belongs to a particular pipeline stage

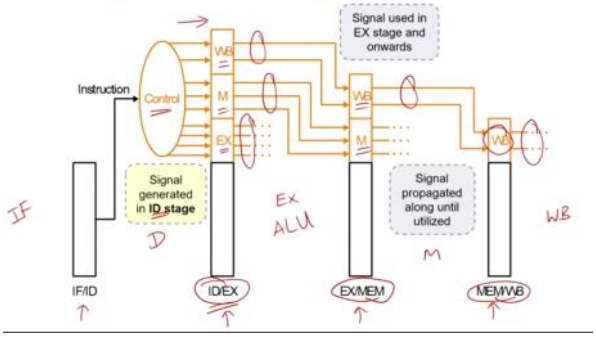
## 4. Pipeline Control: **Grouping**

- Group control signals according to pipeline stage

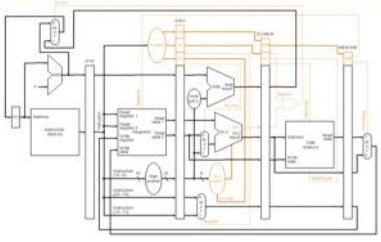
| | RegDst | ALUSrc | MemTo Reg | Reg Write | Mem Read | Mem Write | Branch | ALUop | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | op1 | op0 |
| R-type | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| lw | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| sw | X | 1 | X | 0 | 0 | 1 | 0 | 0 | 0 |
| beq |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

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

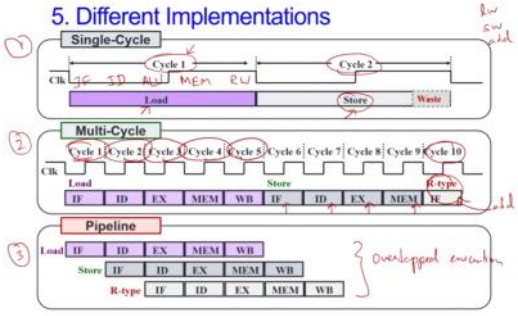
## 4. Pipeline Control: **Implementation**

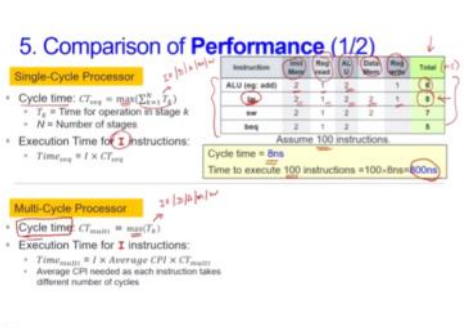


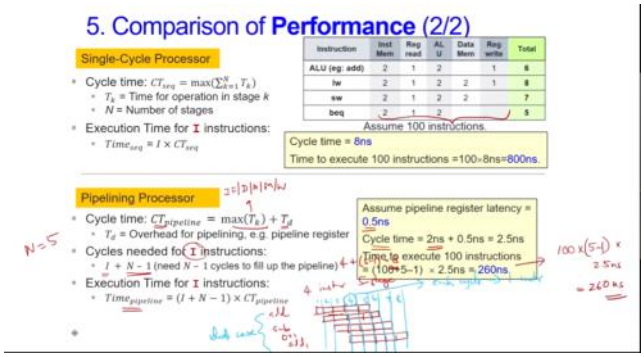

## 4. Pipeline Control: **Datapath** and **Control**



## 5. Different Implementations






## 5. Pipeline Processor: **Ideal Speedup**

- Assumptions for ideal case:
  - Every stage takes the same amount of time → $\sum_{k=1}^{N} T_k = N \times T_1$
  - No pipeline overhead → $T_d = 0$
  - $I >> N$ (Number of instructions is much larger than number of stages)
- $Speedup_{pipeline} = \frac{Time_{seq}}{Time_{pipeline}} = \frac{I \times \sum_{k=1}^{N} T_k}{(I+N-1) \times (\max(T_k)+T_d)} = \frac{I \times N \times T_1}{(I+N-1) \times T_1}$
  $\approx \frac{I \times N \times T_1}{I \times T_1} = N$ ($I >> N$)

**Conclusion:**
Pipeline processor can gain **N** times speedup, where **N** is the number of pipeline stages.

## Review Question

- Given this code:

```
add $t0, $s0, $s1
sub $t1, $s0, $s1
sll $t2, $s0, 2
srl $t3, $s1, 2
```

a) How many cycles will it take to execute the code on a single-cycle datapath?
b) How long will it take to execute the code on a single-cycle datapath, assuming a 100 MHz clock?
c) How many cycles will it take to execute the code on a 5-stage MIPS pipeline?
d) How long will it take to execute the code on a 5-stage MIPS pipeline, assuming a 500 MHz clock?

a) 4 cycles
b) 4/(100×10^6) = 40 ns
c) 4 + 4 = 8 cycles
d) 8/(500×10^6) = 16 ns

1sec = 10^9 ns

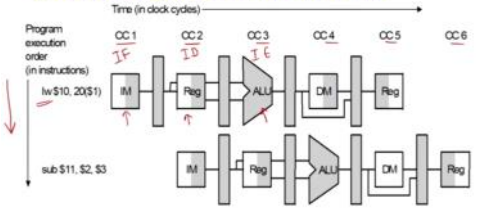
Pipelining can potentially speed up by 5

- Pipeline hazards

## 1. Pipeline Hazards

- Speedup from pipeline implementation:
  - Based on the assumption that a new instructions can be "pumped" into pipeline every cycle
- However, there are **pipeline hazards**
  - Problems that prevent next instruction from immediately following previous instruction
  - **Structural hazards:**
    - Simultaneous use of a hardware resource
  - **Data hazards:**
    - Data dependencies between instructions
  - **Control hazards:**
    - Change in program flow

## 1. Graphical Notation for Pipeline

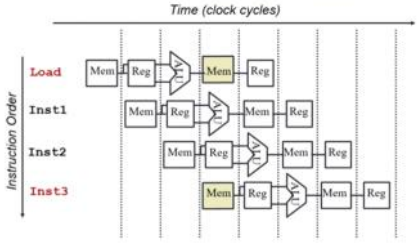



- Horizontal = the stages of an instruction
- Vertical = the instructions in different pipeline stages

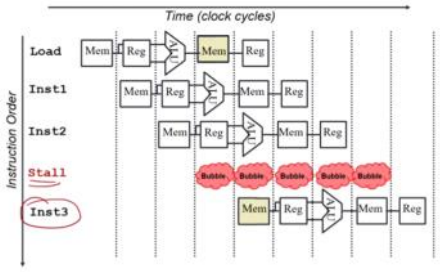
Structural Hazard

## 2. Structural Hazard: Example

- If there is only a **single memory module:**

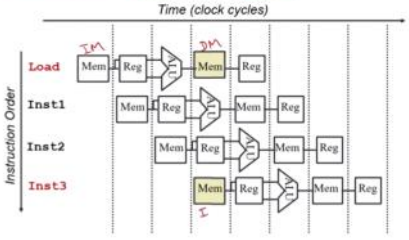



## 2. Solution 1: Stall the Pipeline



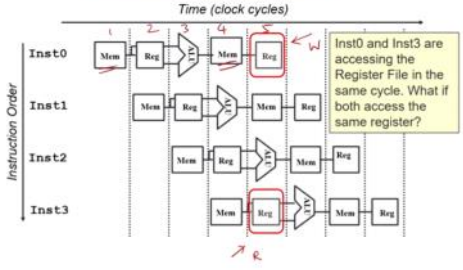

## 2. Solution 2: Separate Memory

- Split memory into **Data** and **Instruction** memory



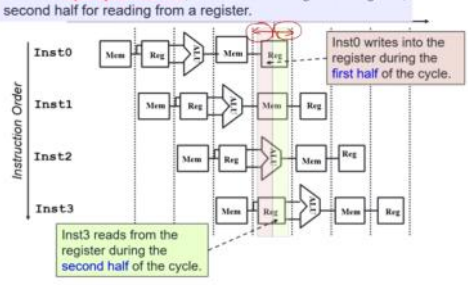

## 2. Quiz (1/2)

Is there another conflict?




## 2. Quiz (2/2)

Recall that registers are very fast memory.
Solution: Split cycle into half; first half for writing into a register; second half for reading from a register.




Data dependency

## 3. Instruction Dependencies

- Instructions can have relationship that prevent pipeline execution:
  - Although a partial overlap maybe possible in some cases
- When different instructions accesses (read/write) the same register
  - Register contention is the cause of dependency
  - Known as **data dependency**
- When the execution of an instruction depends on another instruction
  - Control flow is the cause of dependency
  - Known as **control dependency**
- Failure to handle dependencies can affect **program correctness**!

## 3. Data Dependency: RAW

- "**R**ead-**A**fter-**W**rite" **Definition:**
  - Occurs when a later instruction **reads** from the destination register **written** by an earlier instruction
  - Also known as **true data dependency**

```
i1: add $1, $2, $3 #writes to $1
i2: sub $4, $1, $5 #reads from $1
```

- Effect of incorrect execution:
  - If `i2` reads register `$1` before `i1` can write back the result, `i2` will get a ***stale result (old result)***

## 3. Other Data Dependencies

- Similarly, we have:
  - **WAR**: **W**rite-**a**fter-**R**ead dependency ✗
  - **WAW**: **W**rite-after-**W**rite dependency
- Fortunately, these dependencies **do not cause any pipeline hazards**
- They affect the processor only when instructions are executed out of program order:
  - i.e. in Modern SuperScalar Processor

## 4. RAW Dependency: Hazards?

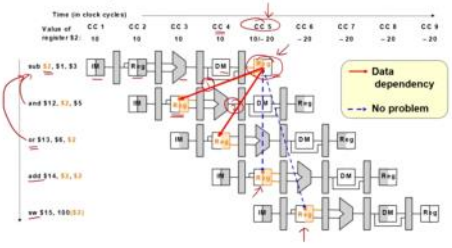
- Suppose we are executing the following code fragment:

```
sub $2, $1, $3    #i1
and $12, $2, $5   #i2
or  $13, $6, $2   #i3
add $14, $2, $2   #i4
sw  $15, 100($2)  #i5
```

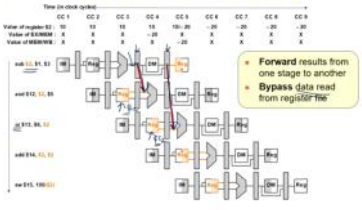
- Note the multiple uses of register $2
- Question:
  - Which are the instructions require special handling?

## 4. RAW Data Hazards

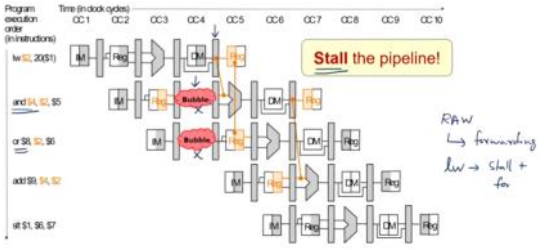
- Value from prior instruction is needed before write back




## 4.1 RAW Data Hazards: **Forwarding**




## 4.2 Data Hazards: **LOAD** Instruction Solution



## 4.3 Exercise #1

- How many cycles will it take to execute the following code on a 5-stage pipeline
  - **without** forwarding?
  - **with** forwarding?

```
sub $2,  $1, $3
and $12, $2, $5
or  $13, $6, $2
add $14, $2, $2
sw  $15, 100($2)
```

## 4.3 Exercise #1: Without Forwarding → 11-cycles

```
sub $2,  $1, $3
and $12, $2, $5
or  $13, $6, $2
add $14, $2, $2
sw  $15, 100($2)
```

| Clock → | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| sub | IF | ID | EX | MEM | WB | | | | | | |
| and | | IF | | | ID | EX | MEM | WB | | | |
| or | | | | | IF | ID | EX | MEM | WB | | |
| add | | | | | | IF | ID | EX | MEM | WB | |
| sw | | | | | | | IF | ID | EX | MEM | WB |

## 4.3 Exercise #1: With Forwarding →

```
sub $2,  $1, $3
and $12, $2, $5
or  $13, $6, $2
add $14, $2, $2
sw  $15, 100($2)
```

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| sub | IF | ID | EX | MEM | WB | | | | | | |
| and | | IF | ID | EX | MEM | WB | | | | | |
| or | | | IF | ID | EX | MEM | WB | | | | |
| add | | | | IF | ID | EX | MEM | WB | | | |
| sw | | | | | IF | ID | EX | MEM | WB | | |

## 4.3 Exercise #2

- How many cycles will it take to execute the following code on a 5-stage pipeline
  - **without** forwarding?
  - **with** forwarding?

```
lw  $2,  20($3)
and $12, $2, $5
or  $13, $6, $2
add $14, $2, $2
sw  $15, 100($2)
```

## 4.3 Exercise #2: Without Forwarding → 11-c cycles

```
lw  $2,  20($3)
and $12, $2, $5
or  $13, $6, $2
add $14, $2, $2
sw  $15, 100($2)
```

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| lw | IF | ID | EX | MEM | WB | | | | | | |
| and | | IF | | | ID | EX | MEM | WB | | | |
| or | | | | | IF | ID | EX | MEM | WB | | |
| add | | | | | | IF | ID | EX | MEM | WB | |
| sw | | | | | | | IF | ID | EX | MEM | WB |

## 4.3 Exercise #2: With Forwarding → 10 c cycles

```
lw  $2,  20($3)
and $12, $2, $5
or  $13, $6, $2
add $14, $2, $2
sw  $15, 100($2)
```

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| lw | IF | ID | EX | MEM | WB | | | | | | |
| and | | IF | ID | | EX | MEM | WB | | | | |
| or | | | IF | | ID | EX | MEM | WB | | | |
| add | | | | | IF | ID | EX | MEM | WB | | |
| sw | | | | | | IF | ID | EX | MEM | WB | |

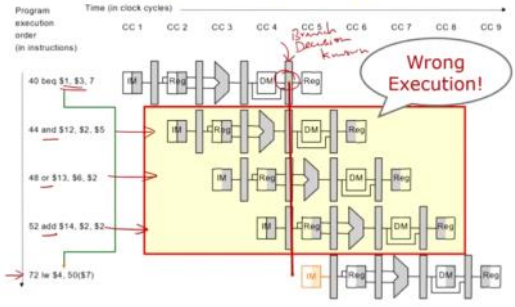
## 5. Control Dependency

- **Definition:**
  - An instruction `j` is control dependent on `i` if `i` controls whether or not `j` executes
  - Typically `i` would be a branch instruction
- Example:

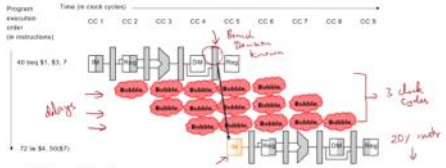
```
i1: beq $3, $5, label   # branch
i2: add $1, $2, $4      # depends on i1
...  ...  ...
```

- Effect of incorrect execution:
  - If `i2` is allowed to execute before `i1` is determined, register `$1` maybe incorrectly changed!

## 5. Control Dependency: **Example**




## 6. Control Hazards: **Stall Pipeline?**




- Wait until the branch outcome is known and then fetch the correct instructions

→ Introduces **3 clock cycles delay**

## 6. Control Hazards: **Reducing the Penalty**

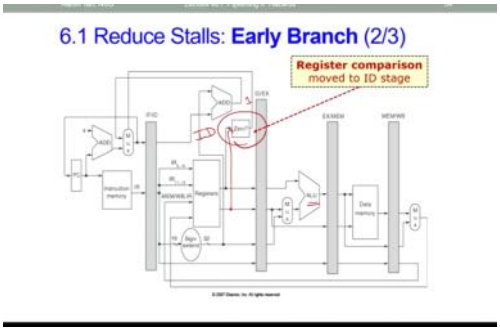
- Branching is very common in code:
  - A 3-cycle stall penalty is too heavy!
- Many techniques invented to reduce the control hazard penalty:
  - Move branch decision calculation to earlier pipeline stage
    - **Early Branch Resolution**
  - Guess the outcome before it is produced
    - **Branch Prediction**
  - Do something useful while waiting for the outcome
    - **Delayed Branching**

Early branch resolution

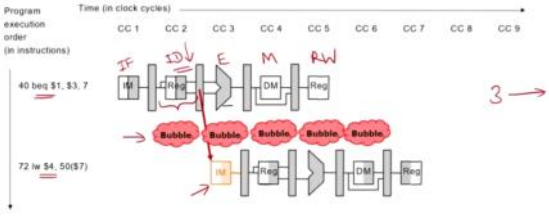
## 6.1 Reduce Stalls: **Early Branch** (1/3)

- Make decision in **ID** stage instead of **MEM**
  - Move branch target address calculation
  - Move register comparison → cannot use ALU for register comparison any more


## 6.1 Reduce Stalls: **Early Branch** (2/3)




## 6.1 Reduce Stalls: **Early Branch** (3/3)

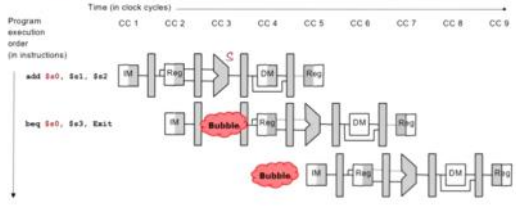



3 → 1 cc

- Wait until the branch decision is known:
  - Then fetch the correct instruction
- Reduced from 3 to **1 clock cycle delay**

## 6.1 Early Branch: **Problems** (2/3)

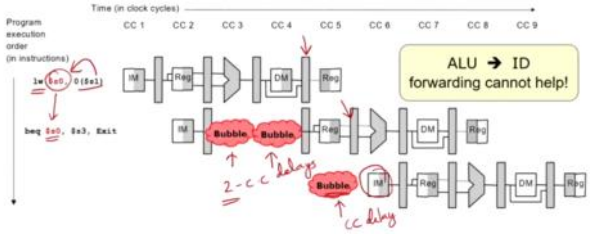
- **Solution:**
  - Add forwarding path from ALU to ID stage
  - **One clock cycle delay** is still needed




<- problem Is if s0 is produced in previous instruction. Since then s0 will not be ready by the time beq requires s0

## 6.1 Early Branch: **Problems** (3/3)

- Problem is worse with **load** followed by **branch**
- **Solution:**
  - MEM to ID forwarding and 2 more stall cycles!
  - In this case, we ended up with 3 total stall cycles → no improvement!



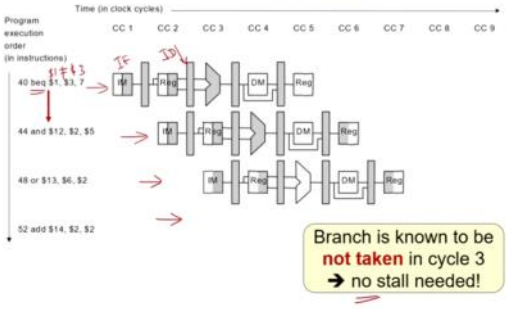

Branch Prediction

-branch is not taken

## 6.2 Reduce Stalls: Branch Prediction

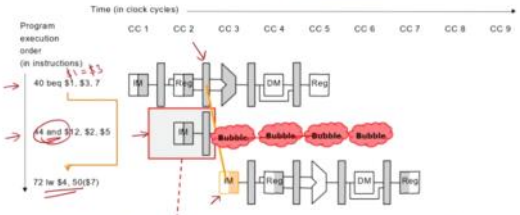
- There are many branch prediction schemes
  - We only cover the simplest in this course ☺
- Simple prediction:
  - All branches are assumed to be **not taken**
  - → Fetch the successor instruction and start pumping it through the pipeline stages
- When the actual branch outcome is known:
  - **Not taken**: Guessed correctly → No pipeline stall
  - **Taken**: Guessed wrongly → Wrong instructions in the pipeline → **Flush** successor instruction from the pipeline

## 6.2 Branch Prediction: Correct Prediction




Branch is known to be **not taken** in cycle 3
→ no stall needed!

## 6.2 Branch Prediction: Wrong Prediction




Branch is known to be **taken** in cycle 3
→ "and" instruction should not be executed
→ Flushed from pipeline

## 6.2 Exercise #3: Branch Prediction

- How many cycles will it take to execute the following code on a 5-stage pipeline **with** forwarding and …
  - **without** branch prediction?
  - **with** branch prediction (predict not taken)?

```
      addi $s0, $zero, 10
Loop: addi $s0, $s0, -1
      bne  $s0, $zero, Loop
      sub  $t0, $t1, $t2
```

- Decision making moved to **ID** stage
- Total instructions = 1 + 10×2 + 1 = 22
- **Ideal** pipeline = 4 + 22 = 26 cycles

26 = (22 instructions- 1) + 5 cycles

## 6.2 Exercise #3: Without Branch Prediction

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| addi¹ | IF | ID | EX | MEM | WB | | | | | | |
| addi² | | IF | ID | EX | MEM | WB | | | | | |
| bne | | | IF | | ID | EX | MEM | WB | | | |
| addi² | | | | | IF | ID | EX | MEM | WB | | |

- Data dependency between (`addi $s0, $s0, -1`) and `bne` incurs 1 cycle of delay. There are 10 iterations, hence 10 cycles of delay.
- Every `bne` incurs a cycle of delay to execute the next instruction. There are 10 iterations, hence 10 cycles of delay.
- Total number of cycles of delay = 20.
- Total execution cycles = 26 + 20 = **46 cycles.**

## 6.2 Exercise #3: With Branch Prediction

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| addi¹ | IF | ID | EX | MEM | WB | | | | | | |
| addi² | | IF | ID | EX | MEM | WB | | | | | |
| bne | | | IF | | ID | EX | MEM | WB | | | |
| sub | | | | | IF | | | | | | |
| addi² | | | | | | IF | ID | EX | MEM | WB | |

**Predict not taken.**

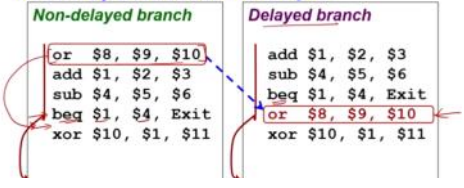
- The data dependency remains, hence 10 cycles of delay for 10 iterations.
- In the first 9 iterations, the branch prediction is wrong, hence 1 cycle of delay.
- In the last iteration, the branch prediction is correct, hence saving 1 cycle of delay.
- Total number of cycles of delay = 19.
- Total execution cycles = 26 + 19 = **45 cycles.**

Delayed branch

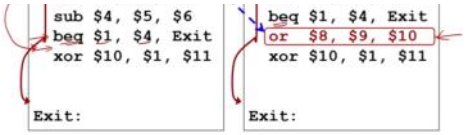
## 6.3 Reduce Stalls: Delayed Branch

- **Observation:**
  - Branch outcome takes **X** number of cycles to be known
  - → **X** cycles stall
- **Idea:**
  - Move **non-control dependent instructions** into the X slots following a branch
    - Known as the **branch-delay slot**

## 6.3 Delayed Branch: Example



- **Idea:**
  - Move **non-control dependent instructions** into the X slots following a branch
    - Known as the **branch-delay slot**
  - → These instructions are executed **regardless of the branch outcome**
- In our MIPS processor:
  - Branch-Delay slot = **1** (with the early branch)




- The "`or`" instruction is moved into the delayed slot:
  - Get executed regardless of the branch outcome
  - → Same behavior as the original code!

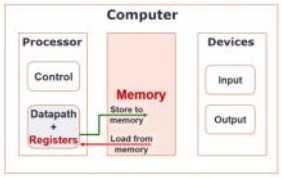
## 6.3 Delayed Branch: **Observation**

- **Best case scenario**
  - There is an instruction **preceding the branch** which **can be moved** into the delayed slot
    - Program correctness must be preserved!
- **Worst case scenario**
  - Such instruction cannot be found
  - → Add a no-op (**nop**) instruction in the branch-delay slot
- Re-ordering instructions is a common method of program optimization
  - Compiler must be smart enough to do this
  - Usually can find such an instruction at least 50% of the time

Week 12

Monday, November 13, 2023 7:13 PM

## 1. Data Transfer: The Big Picture

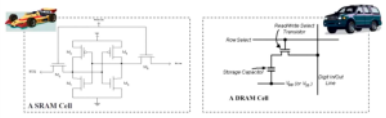



Registers are in the datapath of the processor. If operands are in memory we have to **load** them to processor (registers), operate on them, and **store** them back to memory.

## 1. Memory Technology Today: DRAM

- **DDR SDRAM**
  - **D**ouble **D**ata **R**ate
    - – **S**ynchronous **D**ynamic **RAM**
  - The dominant memory technology in PC market
  - Delivers memory on the positive and negative edge of a clock (double rate)
  - Generations:
    - DDR (MemClkFreq x 2(double rate) x 8 words)
    - DDR2 (MemClkFreq x 2(multiplier) x 2 x 8 words)
    - DDR3 (MemClkFreq x 4(multiplier) x 2 x 8 words)
    - DDR4 (released in 2014)
    - DDR5 (in Q3 2021)

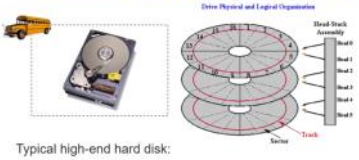
## 1. Faster Memory Technology: SRAM



**SRAM**
6 transistors per memory cell
→ **Low density**
**Fast access** latency of 0.5 – 5 ns
More costly
Uses flip-flops

**DRAM**
1 transistor per memory cell
→ **High density**
**Slow access** latency of 50-70ns
Less costly
Used in main memory

## 1. Slow Memory Technology: Magnetic Disk



Typical high-end hard disk:

Average Latency: 4 - 10 ms
Capacity: 500-2000GB

| | Capacity | Latency | Cost/GB |
|---|---|---|---|
| Register | 100s Bytes | 20 ps | $$$$ |
| SRAM | 100s KB | 0.5-5 ns | $$$ |
| DRAM | 100s MB | 50-70 ns | $ |
| Hard Disk | 100s GB | 5-20 ms | Cents |
| **Ideal** | **1 GB** | **1 ns** | **Cheap** |

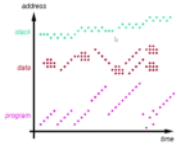
## 2. Cache: The Basic Idea

- Keep the frequently and recently used data in **smaller but faster** memory
- Refer to bigger and slower memory:
  - Only when you cannot find data/instruction in the faster memory
- Why does it work?

**Principle of Locality**
Program accesses only a small portion of the memory address space within a small time interval

## 2.1 Cache: Types of Locality

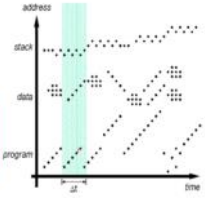
- **Temporal locality**
  - If an item is referenced, it will tend to be referenced again soon
- **Spatial locality**
  - If an item is referenced, nearby items will tend to be referenced soon
- Different locality for
  - Instructions
  - Data



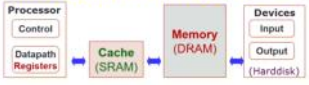

## 2.1 Working Set: Definition

- **Set of locations accessed during Δt**
- Different phases of execution may use different working sets

Our aim is to **capture the working set and keep it in the memory closest to CPU**

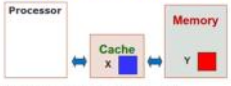



## 2.2 Two Aspects of Memory Access




- How to make SLOW main memory appear faster?
  - **Cache** – a small but fast SRAM near CPU
  - **Hardware managed:** Transparent to programmer
- How to make SMALL main memory appear bigger than it is?
  - **Virtual memory**
  - **OS managed:** Transparent to programmer
  - Not in the scope of this module (covered in CS2106)

## 2.2 Memory Access Time: Terminology




- **Hit**: Data is in cache (e.g., **X**)
  - Hit rate: Fraction of memory accesses that hit
  - Hit time: Time to access cache
- **Miss**: Data is not in cache (e.g., **Y**)
  - Miss rate = 1 – Hit rate
  - Miss penalty: Time to replace cache block + hit time
- Hit time < Miss penalty

## 2.2 Memory Access Time: Formula

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

Example:

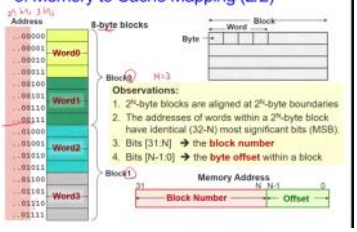
- Suppose our on-chip SRAM (cache) has **0.8 ns** access time, but the fastest DRAM (main memory) we can get has an access time of **10ns**. **How high a hit rate** do we need to sustain an average access time of **1ns**?

Let $h$ be the desired hit rate.
$1 = 0.8h + (1 - h) \times (10 + 0.8)$
$= 0.8h + 10.8 - 10.8h$
$10h = 9.8 \rightarrow h = 0.98$
Hence we need a hit rate of **98%**.

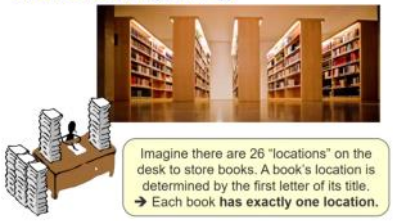
## 3. Memory to Cache Mapping (1/2)

- **Cache Block/Line:**
  - Unit of transfer between memory and cache
- Block size is typically one or more words
  - e.g.: 16-byte block = 4-word block
  - 32-byte block = 8-word block
- Why is the block size bigger than word size?

## 3. Memory to Cache Mapping (2/2)

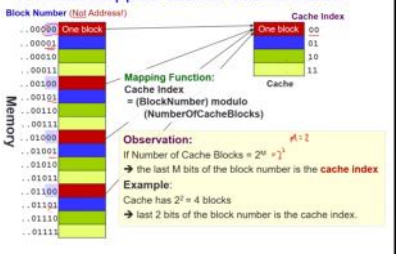



## 4. Direct Mapping Analogy

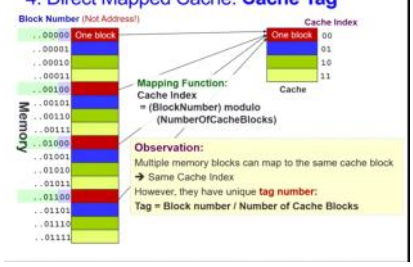


Imagine there are 26 "locations" on the desk to store books. A book's location is determined by the first letter of its title.
→ Each book **has exactly one location.**

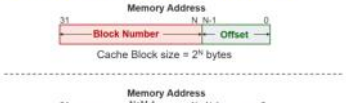
## 4. Direct Mapped Cache: **Cache Index**



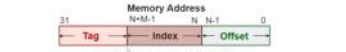

## 4. Direct Mapped Cache: **Cache Tag**




## 4. Direct Mapped Cache: **Mapping**

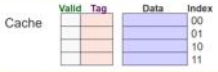



Cache Block size = $2^N$ bytes




Cache Block size = $2^N$ bytes
Number of cache blocks = $2^M$
**Offset** = **N bits**
**Index** = **M bits**
**Tag** = **32 – (N + M) bits**

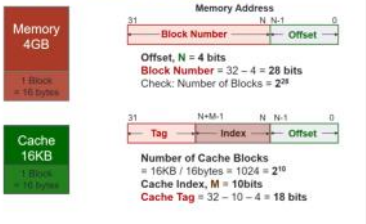
## 4. Direct Mapped Cache: **Cache Structure**




Along with a data block (line), cache also contains the following administrative information (overheads):
1. **Tag** of the memory block
2. **Valid bit** indicating whether the cache line contains valid data

**When is there a cache hit?**
( Valid[index] = TRUE ) **AND**
( Tag[ index ] = Tag[ memory address ] )

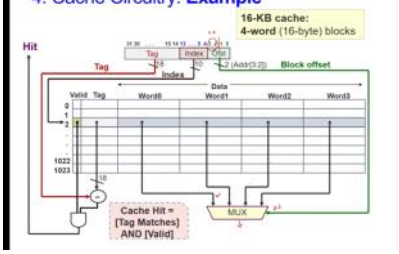
## 4. Cache Mapping: **Example**




**Offset, N = 4 bits**
**Block Number** = 32 – 4 = **28 bits**
Check: Number of Blocks = $2^{28}$

**Number of Cache Blocks**
= 16KB / 16bytes = 1024 = $2^{10}$
**Cache Index, M = 10bits**
**Cache Tag** = 32 – 10 – 4 = **18 bits**

## 4. Cache Circuitry: **Example**




## 5. Reading Data: **Setup**

- Given a direct mapped 16KB cache:
  - 16-byte blocks x 1024 cache blocks
- Trace the following memory accesses:

| Tag (31–14) | Index (13–4) | Offset (3–0) |
|---|---|---|
| 00000000000000000 | 0000000001 | 0100 |
| 00000000000000000 | 0000000001 | 1100 |
| 00000000000000000 | 0000000011 | 0100 |
| 00000000000000010 | 0000000001 | 1000 |
| 00000000000000000 | 0000000001 | 0000 |

## 5. Reading Data: **Load #1-2**

- Load from Tag: 00000000000000000 | Index: 0000000001 | Offset: 0100

**Step 2**. Data in block 1 is **invalid [Cold/Compulsory Miss]**

| Index | Valid | Tag | Word0 Bytes 0-3 | Word1 Bytes 4-7 | Word2 Bytes 8-11 | Word3 Bytes 12-15 |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 0 | | | | | |
| 2 | 0 | | | | | |
| 3 | 0 | | | | | |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

## 5. Reading Data: **Load #1-3**

- Load from Tag: 00000000000000000 | Index: 0000000001 | Offset: 0100

**Step 3**. Load 16 bytes from memory; Set **Tag** and **Valid** bit

| Index | Valid | Tag | Word0 Bytes 0-3 | Word1 Bytes 4-7 | Word2 Bytes 8-11 | Word3 Bytes 12-15 |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 0 | A | B | C | D |
| 2 | 0 | | | | | |
| 3 | 0 | | | | | |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

## 5. Reading Data: **Load #1-4**

- Load from Tag: 00000000000000000 | Index: 0000000001 | Offset: 0100

**Step 4**. Return **Word1** (byte offset = 4) to Register

| Index | Valid | Tag | Word0 Bytes 0-3 | Word1 Bytes 4-7 | Word2 Bytes 8-11 | Word3 Bytes 12-15 |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 0 | A | B | C | D |
| 2 | 0 | | | | | |
| 3 | 0 | | | | | |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

## 5. Reading Data: **Load #2-1**

- Load from Tag: 00000000000000000 | Index: 0000000001 | Offset: 1100

**Step 1**. Check Cache Block at index **1**

| Index | Valid | Tag | Word0 Bytes 0-3 | Word1 Bytes 4-7 | Word2 Bytes 8-11 | Word3 Bytes 12-15 |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 0 | A | B | C | D |
| 2 | 0 | | | | | |
| 3 | 0 | | | | | |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

## 5. Reading Data: **Load #2-2**

- Load from Tag: 00000000000000000 | Index: 0000000001 | Offset: 1100

**Step 2**. [Cache Block is Valid] AND [Tags match] ➔ Cache hit!

| Index | Valid | Tag | Word0 Bytes 0-3 | Word1 Bytes 4-7 | Word2 Bytes 8-11 | Word3 Bytes 12-15 |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 0 | A | B | C | D |
| 2 | 0 | | | | | |
| 3 | 0 | | | | | |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

## 5. Reading Data: **Load #2-3**

- Load from Tag: 00000000000000000 | Index: 0000000001 | Offset: 1100

**Step 3**. Return **Word3** (byte offset = 12) to Register **[Spatial Locality]**

| Index | Valid | Tag | Word0 Bytes 0-3 | Word1 Bytes 4-7 | Word2 Bytes 8-11 | Word3 Bytes 12-15 |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 0 | A | B | C | D |
| 2 | 0 | | | | | |
| 3 | 0 | | | | | |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

## 5. Reading Data: **Load #3-1**

- Load from Tag: 00000000000000000 | Index: 0000000011 | Offset: 0100

**Step 1**. Check Cache Block at index **3**

| Index | Valid | Tag | Word0 Bytes 0-3 | Word1 Bytes 4-7 | Word2 Bytes 8-11 | Word3 Bytes 12-15 |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 0 | A | B | C | D |
| 2 | 0 | | | | | |
| 3 | 0 | | | | | |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

## 5. Reading Data: **Load #3-3**

- Load from Tag: 00000000000000000 | Index: 0000000011 | Offset: 0100

**Step 3**. Load 16 bytes from memory; Set **Tag** and **Valid** bit

| Index | Valid | Tag | Word0 Bytes 0-3 | Word1 Bytes 4-7 | Word2 Bytes 8-11 | Word3 Bytes 12-15 |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 0 | A | B | C | D |
| 2 | 0 | | | | | |
| 3 | 1 | 0 | I | J | K | L |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

## 5. Reading Data: **Load #3-4**

- Load from Tag: 00000000000000000 | Index: 0000000011 | Offset: 0100

**Step 4**. Return **Word1** (byte offset = 4) to Register

| Index | Valid | Tag | Word0 Bytes 0-3 | Word1 Bytes 4-7 | Word2 Bytes 8-11 | Word3 Bytes 12-15 |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 0 | A | B | C | D |
| 2 | 0 | | | | | |
| 3 | 1 | 0 | I | J | K | L |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

## 5. Reading Data: **Load #4-1**

- Load from Tag: 00000000000000010 | Index: 0000000001 | Offset: 1000

**Step 1**. Check Cache Block at index **1**

| Index | Valid | Tag | Word0 Bytes 0-3 | Word1 Bytes 4-7 | Word2 Bytes 8-11 | Word3 Bytes 12-15 |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 0 | A | B | C | D |
| 2 | 0 | | | | | |
| 3 | 1 | 0 | I | J | K | L |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

## 5. Reading Data: **Load #4-2**

- Load from Tag: 00000000000000010 | Index: 0000000001 | Offset: 1000

**Step 2**. Cache block is **Valid** but **Tags mismatch [Cold miss]**

| Index | Valid | Tag | Word0 Bytes 0-3 | Word1 Bytes 4-7 | Word2 Bytes 8-11 | Word3 Bytes 12-15 |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 0 | A | B | C | D |
| 2 | 0 | | | | | |
| 3 | 1 | 0 | I | J | K | L |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

## 5. Reading Data: **Load #4-3**

- Load from Tag: 00000000000000010 | Index: 0000000001 | Offset: 1000

**Step 3**. Replace block **1** with new data; Set **Tag**

| Index | Valid | Tag | Word0 Bytes 0-3 | Word1 Bytes 4-7 | Word2 Bytes 8-11 | Word3 Bytes 12-15 |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 2 | E | F | G | H |
| 2 | 0 | | | | | |
| 3 | 1 | 0 | I | J | K | L |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

## 5. Reading Data: Load #5-1

- Load from 00000000000000000000 0000000001 0000

**Step 1**. Check Cache Block at index **1**

| Index | Valid | Tag | Word0 Bytes 0-3 | Word1 Bytes 4-7 | Word2 Bytes 8-11 | Word3 Bytes 12-15 |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 2 | E | F | G | H |
| 2 | 0 | | | | | |
| 3 | 1 | 0 | I | J | K | L |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| ... | | | | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

## 5. Reading Data: Load #5-2

- Load from 00000000000000000000 0000000001 0000

**Step 2**. Cache block is **Valid** but **Tags mismatch** [Cold miss]

| Index | Valid | Tag | Word0 Bytes 0-3 | Word1 Bytes 4-7 | Word2 Bytes 8-11 | Word3 Bytes 12-15 |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 2 | E | F | G | H |
| 2 | 0 | | | | | |
| 3 | 1 | 0 | I | J | K | L |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| ... | | | | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

## 5. Reading Data: Load #5-3

- Load from 00000000000000000000 0000000001 0000

**Step 3**. Replace block 1 with new data; Set **Tag**

| Index | Valid | Tag | Word0 Bytes 0-3 | Word1 Bytes 4-7 | Word2 Bytes 8-11 | Word3 Bytes 12-15 |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 0 | A | B | C | D |
| 2 | 0 | | | | | |
| 3 | 1 | 0 | I | J | K | L |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| ... | | | | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

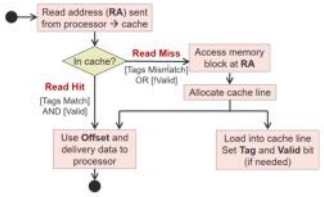
## 5. Reading Data: Load #5-4

- Load from 00000000000000000000 0000000001 0000

**Step 4**. Return **Word0** (byte offset = 0) to Register

| Index | Valid | Tag | Word0 Bytes 0-3 | Word1 Bytes 4-7 | Word2 Bytes 8-11 | Word3 Bytes 12-15 |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 0 | A | B | C | D |
| 2 | 0 | | | | | |
| 3 | 1 | 0 | I | J | K | L |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| ... | | | | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

## 5. Reading Data: Summary




## 6. Types of Cache Misses

- Compulsory misses
  - On the first access to a block; the block must be brought into the cache
  - Also called cold start misses or first reference misses
- Conflict misses
  - Occur in the case of direct mapped cache or set associative cache, when several blocks are mapped to the same block/set
  - Also called collision misses or interference misses
- Capacity misses
  - Occur when blocks are discarded from cache as cache cannot contain all blocks needed

## Exercise #1: Setup Information

Memory 4GB — 1 Block = 8 bytes

Cache 32Bytes — 1 Block = 8 bytes

Memory Address

31 ... N N-1 ... 0 | Block Number | Offset

Offset, N = 3

Block Number = 29

31 ... N+M-1 N N-1 ... 0 | Tag | Index | Offset

Number of Cache Blocks = 4

Cache Index, M = 2

Cache Tag = 27

## Exercise #2: Tracing Memory Accesses

- Using the given setup in exercise #1, trace the following memory loads:
  - Load from addresses:
    **4, 0, 8, 12, 36, 0, 4**
- Note that "A", "B"…. "J" represent word-size data
  - Assume 1 word = 4 bytes

**Memory Content**

| Addr | Data |
|---|---|
| 0 | A |
| 4 | B |
| 8 | C |
| 12 | D |
| ... | ... |
| 32 | I |
| 36 | J |
| ... | ... |

## Exercise #2: Load #1

Addresses: 4, 0, 8, 12, 36, 0, 4

Address 4 = 00000000000000000000000000 | 00 | 100 (Tag | Index | Offset)

| Index | Valid | Tag | Word0 | Word1 |
|---|---|---|---|---|
| 0 | 0 → 1 | 0 | A | B |
| 1 | 0 | | | |
| 2 | 0 | | | |
| 3 | 0 | | | |

## Exercise #2: Load #2

Addresses: 4, 0, 8, 12, 36, 0, 4 (Miss Hit)

Address 0 = 00000000000000000000000000 | 00 | 000 (Tag | Index | Offset)

| Index | Valid | Tag | Word0 | Word1 |
|---|---|---|---|---|
| 0 | 1 | 0 | A | B |
| 1 | 0 | | | |
| 2 | 0 | | | |
| 3 | 0 | | | |

## Exercise #2: Load #3

Addresses: 4, 0, 8, 12, 36, 0, 4 (Miss Hit Miss)

Address 8 = 00000000000000000000000000 | 01 | 000 (Tag | Index | Offset)

| Index | Valid | Tag | Word0 | Word1 |
|---|---|---|---|---|
| 0 | 1 | 0 | A | B |
| 1 | 0 → 1 | 0 | C | D |
| 2 | 0 | | | |
| 3 | 0 | | | |

## Exercise #2: Load #5

Addresses: 4, 0, 8, 12, 36, 0, 4 (Miss Hit Miss Hit Miss)

Address 36 = 00000000000000000000000001 | 00 | 100 (Tag | Index | Offset)

| Index | Valid | Tag | Word0 | Word1 |
|---|---|---|---|---|
| 0 | 1 | 0 → 1 | A → I | B → J |
| 1 | 1 | 0 | C | D |
| 2 | 0 | | | |
| 3 | 0 | | | |

## Exercise #2: Load #7

Addresses: 4, 0, 8, 12, 36, 0, 4 (Miss Hit Miss Hit Miss Miss Hit)

Address 4 = 00000000000000000000000000 | 00 | 100 (Tag | Index | Offset)

| Index | Valid | Tag | Word0 | Word1 |
|---|---|---|---|---|
| 0 | 1 | 0 → 0 | A → A | B → B |
| 1 | 1 | 0 | C | D |
| 2 | 0 | | | |
| 3 | 0 | | | |

Writing data

## 7. Writing Data: Store #1-1

- Store **X** to 00000000000000000000 0000000001 1000 (Tag | Index | Offset)

**Step 1**. Check Cache Block **1**

| Index | Valid | Tag | Word0 Bytes 0-3 | Word1 Bytes 4-7 | Word2 Bytes 8-11 | Word3 Bytes 12-15 |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |

## 7. Writing Data: Store #1-2

- Store **X** to 00000000000000000000 0000000001 1000 (Tag | Index | Offset)

**Step 2**. [Cache Block is Valid] AND [Tags match] → Cache hit!

| Index | Valid | Tag | Word0 Bytes 0-3 | Word1 Bytes 4-7 | Word2 Bytes 8-11 | Word3 Bytes 12-15 |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 0 | A | B | C | D |

## 7. Writing Data: Store #1-3

- Store **X** to 00000000000000000000 0000000001 1000 (Tag | Index | Offset)

**Step 2**. Replace Word2 (offset = 8) with **X**

| Index | Valid | Tag | Word0 Bytes 0-3 | Word1 Bytes 4-7 | Word2 Bytes 8-11 | Word3 Bytes 12-15 |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 0 | A | B | X | D |

- Store **X** to 0000000000000000 0000000001 1000

**Step 1**. Check Cache Block **1**

| Index | Valid | Tag | Word0 Bytes 0-3 | Word1 Bytes 4-7 | Word2 Bytes 8-11 | Word3 Bytes 12-15 |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 0 | A | B | C | D |
| 2 | 0 | | | | | |
| 3 | 1 | 0 | I | J | K | L |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| ... | | | | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

- Store **X** to 0000000000000000 0000000001 1000

**Step 2**. [Cache Block is Valid] AND [Tags match] → Cache hit!

| Index | Valid | Tag | Word0 Bytes 0-3 | Word1 Bytes 4-7 | Word2 Bytes 8-11 | Word3 Bytes 12-15 |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 0 | A | B | C | D |
| 2 | 0 | | | | | |
| 3 | 1 | 0 | I | J | K | L |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| ... | | | | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

- Store **X** to 0000000000000000 0000000001 1000

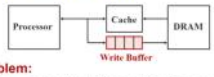
**Step 2**. Replace Word2 (offset = 8) with **X**

| Index | Valid | Tag | Word0 Bytes 0-3 | Word1 Bytes 4-7 | Word2 Bytes 8-11 | Word3 Bytes 12-15 |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 0 | A | B | X | D |
| 2 | 0 | | | | | |
| 3 | 1 | 0 | I | J | K | L |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| ... | | | | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

## 8. Changing Cache Content: **Write Policy**

- Cache and main memory are inconsistent
  - Modified data only in cache, not in memory!
- **Solution 1: Write-through** cache
  - Write data both to cache and to main memory
- **Solution 2: Write-back** cache
  - Only write to cache
  - Write to main memory only when cache block is replaced (evicted)

## 8. Write-Through Cache




- **Problem:**
  - Write will operate at the speed of main memory!
- **Solution:**
  - Put a write buffer between cache and main memory
    - Processor: writes data to cache + write buffer
    - Memory controller: write contents of the buffer to memory

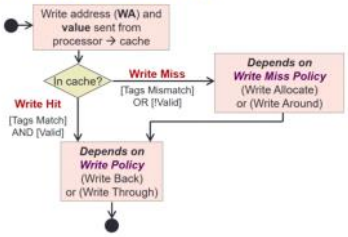
## 8. Write-Back Cache

- **Problem:**
  - Quite wasteful if we write back every evicted cache blocks
- **Solution:**
  - Add an additional bit (**Dirty bit**) to each cache block
  - Write operation will change dirty bit to 1
    - Only cache block is updated, no write to memory
  - When a cache block is replaced:
    - Only write back to memory if dirty bit is 1

## 8. Handling Cache Misses

- On a **Read Miss**:
  - Data loaded into cache and then load from there to register
- **Write Miss** option 1: **Write allocate**
  - Load the complete block into cache
  - Change only the required word in cache
  - Write to main memory depends on write policy
- **Write Miss** option 2: **Write around**
  - Do not load the block to cache
  - Write directly to **main memory only**

## 8. Writing Data: Summary



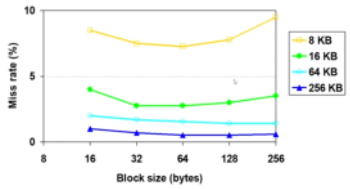

Associative Cache

## 2. Block Size Trade-off (1/2)

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

- Larger block size:
  - \+ Takes advantage of spatial locality
  - − Larger miss penalty: Takes longer time to fill up the block
  - − If block size is too big relative to cache size
    - → Too few cache blocks → miss rate will go up


## 2. Block Size Trade-off (2/2)




## 3. Set Associative (SA) Cache

- Compulsory misses
  - On the first access to a block; the block must be brought into the cache
  - Also called cold start misses or first reference misses
- Conflict misses
  - Occur in the case of direct mapped cache or set associative cache, when several blocks are mapped to the same block/set
  - Also called collision misses or interference misses
- Capacity misses
  - Occur when blocks are discarded from cache as cache cannot contain all blocks needed

## 3. Set Associative (SA) Cache

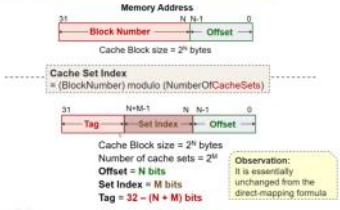
- **N-way Set Associative Cache**
  - A memory block can be placed in a fixed number (**N**) of locations in the cache, where **N** > 1
- **Key Idea:**
  - Cache consists of a number of sets:
    - Each set contains **N** cache blocks
  - Each memory block maps to a unique cache set
  - Within the set, a memory block can be placed in **any** of the **N** cache blocks in the set

## 3. Set Associative Cache: **Structure**

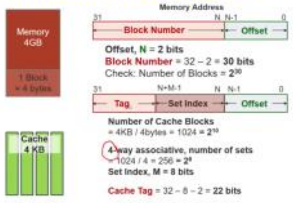

- An example of 2-way set associative cache
  - Each set has two cache blocks
- A memory block maps to a **unique set**
  - In the set, the memory block can be placed in **either of the cache blocks**

→ Need to search both to look for the memory block

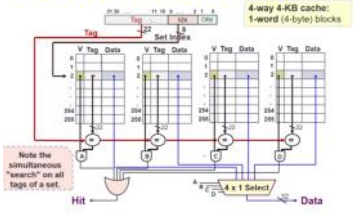
## 3. Set Associative Cache: **Mapping**



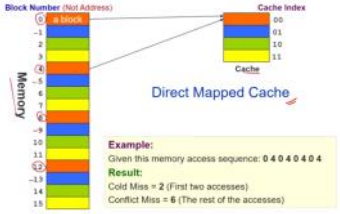

## 3. Set Associative Cache: **Example**



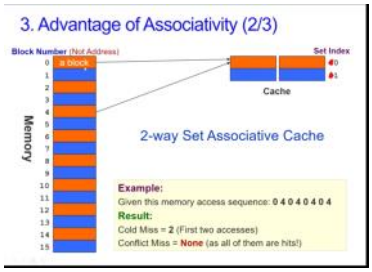

## 3. Set Associative Cache: **Circuitry**



## 3. Advantage of Associativity (1/3)

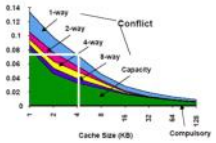



## 3. Advantage of Associativity (2/3)




## 3. Advantage of Associativity (3/3)

**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**




## 3. SA Cache Example: **Setup**

- Given:
  - Memory access sequence: **4, 0, 8, 36, 0**
  - 2-way set-associative cache with a total of four 8-byte blocks → **total of 2 sets**
  - Indicate hit/miss for each access


**Offset, N** = 3 bits
**Block Number** = 32 – 3 = 29 bits

**2-way associative, number of sets** = 2 = $2^1$
**Set Index, M** = 1 bits

**Cache Tag** = 32 – 3 – 1 = **28 bits**

## 3. SA Cache Example: **Load #1**

4, 0, 8, 36, 0

- Load from **4** → 00000000000000000000000000 | 0 | 100 (Tag | Index | Offset)

**Check:** Both blocks in **Set 0** are invalid **[ Cold Miss ]**

**Result:** Load from memory and place in **Set 0 - Block 0**

| Set Index | Block 0 | | | | Block 1 | | | |
|---|---|---|---|---|---|---|---|---|
| | Valid | Tag | W0 | W1 | Valid | Tag | W0 | W1 |
| 0 | 1 | 0 | M[0] | M[4] | 0 | | | |
| 1 | 0 | | | | 0 | | | |

## 3. SA Cache Example: **Load #2**

4, 0, 8, 36, 0

- Load from **0** → 00000000000000000000000000 | 0 | 000 (Tag | Index | Offset)

**Result:**
[Valid and Tags match] in **Set 0-Block 0 [ Spatial Locality ]**

| Set Index | Block 0 | | | | Block 1 | | | |
|---|---|---|---|---|---|---|---|---|
| | Valid | Tag | W0 | W1 | Valid | Tag | W0 | W1 |
| 0 | 1 | 0 | M[0] | M[4] | 0 | | | |
| 1 | 0 | | | | 0 | | | |

## 3. SA Cache Example: **Load #4**

4, 0, 8, 36, 0

- Load from **36** → 00000000000000000000000010 | 0 | 100 (Tag | Index | Offset)

**Check:** [Valid but tag mismatch] **Set 0 - Block 0**
[Invalid] **Set 0 - Block1 [ Cold Miss ]**

| Set Index | Block 0 | | | | Block 1 | | | |
|---|---|---|---|---|---|---|---|---|
| | Valid | Tag | W0 | W1 | Valid | Tag | W0 | W1 |
| 0 | 1 | 0 | M[0] | M[4] | 0 | | | |
| 1 | 1 | 0 | M[8] | M[12] | 0 | | | |

## 3. SA Cache Example: **Load #5**

4, 0, 8, 36, 0

- Load from **0** → 00000000000000000000000000 | 0 | 000 (Tag | Index | Offset)

**Check:** [Valid and tags match] **Set 0-Block 0**
[Valid but tags mismatch] **Set 0-Block1**
**[ Temporal Locality ]**

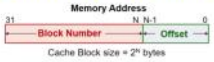
| Set Idx | Block 0 | | | | Block 1 | | | |
|---|---|---|---|---|---|---|---|---|
| | Valid | Tag | W0 | W1 | Valid | Tag | W0 | W1 |
| 0 | 1 | 0 | M[0] | M[4] | 1 | 2 | M[32] | M[36] |
| 1 | 1 | 0 | M[8] | M[12] | 0 | | | |

Fully Associative Cache

## 4. Fully Associative (FA) Cache

- **Fully Associative Cache**
  - A memory block can be placed in any location in the cache
- **Key Idea:**
  - Memory block placement is no longer restricted by cache index or cache set index
  - ++ Can be placed in any location, **BUT**
  - --- Need to search all cache blocks for memory access

## 4. Fully Associative Cache: **Mapping**




Cache Block size = $2^N$ bytes


Cache Block size = $2^N$ bytes
Number of cache blocks = $2^M$
**Offset** = **N bits**
**Tag** = **32 – N bits**

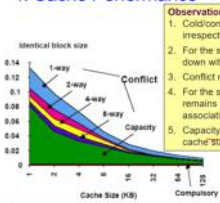
**Observation:**
The block number serves as the tag in FA cache.

## 4. Fully Associative Cache: **Circuitry**

- Example:
  - 4KB cache size and 16-Byte block size
  - Compare tags and valid bit in parallel


No Conflict Miss

## 4. Cache Performance




**Observations:**
1. Cold/compulsory miss remains the same irrespective of cache size/associativity.
2. For the same cache size, conflict miss goes down with increasing associativity.
3. Conflict miss is 0 for FA caches.
4. For the same cache size, capacity miss remains the same irrespective of associativity.
5. Capacity miss decreases with increasing cache size.

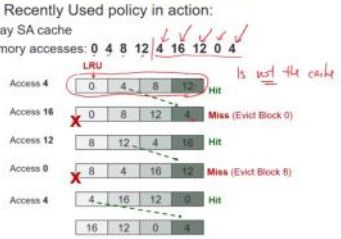
Total Miss = Cold miss + Conflict miss + Capacity miss
**Capacity miss (FA) = Total miss (FA) – Cold miss (FA)**, when Conflict Miss→0

## 5. Block Replacement Policy (1/3)

- Set Associative or Fully Associative Cache:
  - Can choose where to place a memory block
  - Potentially replacing another cache block if full
  - Need **block replacement policy**
- Least Recently Used (**LRU**)
  - **How:** For cache hit, record the cache block that was accessed
    - When replacing a block, choose one which has not been accessed for the longest time
  - **Why:** Temporal locality

## 5. Block Replacement Policy (2/3)

- Least Recently Used policy in action:
  - 4-way SA cache
  - Memory accesses: **0 4 8 12 4 16 12 0 4**




## 5. Block Replacement Policy (3/3)

- Drawback for LRU
  - Hard to keep track if there are many choices
- Other replacement policies:
  - First in first out (FIFO)
  - Random replacement (RR)
  - Least frequently used (LFU)

## 6. Additional Examples #1

- **Direct-Mapped** Cache:
  - Four 8-byte blocks
- Memory accesses:
  **4, 8, 36, 48, 68, 0, 32**

| Addr: | Tag | Index | Offset |
|---|---|---|---|
| 4: | 00…000 | 00 | 100 |
| 8: | 00…000 | 01 | 000 |
| 36: | 00…001 | 00 | 100 |
| 48: | 00…001 | 10 | 000 |
| 68: | 00…010 | 00 | 100 |
| 0: | 00…000 | 00 | 000 |
| 32: | 00…001 | 00 | 000 |

| Index | Valid | Tag | Word0 | Word1 |
|---|---|---|---|---|
| 0 | ~~0~~ 1 | ~~0~~ 1 / ~~1~~ 2 / ~~0~~ 0 | ~~M[0]~~ M[32] / ~~M[32]~~ M[64] / ~~M[0]~~ M[4] | ~~M[4]~~ M[36] / ~~M[36]~~ M[68] |
| 1 | ~~0~~ 1 | 0 | M[8] | M[12] |
| 2 | ~~0~~ 1 | 1 | M[48] | M[52] |
| 3 | 0 | | | |

## 6. Additional Examples #2

- **Fully-Associative** Cache:
  - Four 8-byte blocks
  - LRU Replacement Policy
- Memory accesses:
  **4, 8, 36, 48, 68, 0, 32**

| Addr: | Tag | Offset |
|---|---|---|
| 4: | 00…00000 | 100 |
| 8: | 00…00001 | 000 |
| 36: | 00…00100 | 100 |
| 48: | 00…00110 | 000 |
| 68: | 00…01000 | 100 |
| 0: | 00…00000 | 000 |
| 32: | 00…00100 | 000 |

| Index | Valid | Tag | Word0 | Word1 |
|---|---|---|---|---|
| 0 | ~~0~~ 1 | ~~0~~ 8 | ~~M[0]~~ M[64] | ~~M[4]~~ M[68] |
| 1 | ~~0~~ 1 | ~~1~~ 0 | ~~M[8]~~ M[0] | ~~M[12]~~ M[4] |
| 2 | ~~0~~ 1 | 4 | M[32] | M[36] |
| 3 | ~~0~~ 1 | 6 | M[48] | M[52] |

## 6. Additional Examples #3

- **2-way Set-Associative** Cache:
  - Four 8-byte blocks
  - LRU Replacement Policy
- Memory accesses:
  **4, 8, 36, 48, 68, 0, 32**

| Addr: | Tag | Index | Offset |
|---|---|---|---|
| 4: | 00…0000 | 0 | 100 |
| 8: | 00…0000 | 1 | 000 |
| 36: | 00…0010 | 0 | 100 |
| 48: | 00…0011 | 0 | 000 |
| 68: | 00…0100 | 0 | 100 |
| 0: | 00…0000 | 0 | 000 |
| 32: | 00…0010 | 0 | 000 |

| Set Index | Block 0 | | | | Block 1 | | | |
|---|---|---|---|---|---|---|---|---|
| | Valid | Tag | Word0 | Word1 | Valid | Tag | Word0 | Word1 |
| 0 | ~~0~~ 1 | ~~0~~ 3 / 0 | ~~M[0]~~ M[48] / M[0] | ~~M[4]~~ M[52] / M[4] | ~~0~~ 1 | ~~2~~ 4 / 2 | ~~M[32]~~ M[64] / M[32] | ~~M[36]~~ M[68] / M[36] |
| 1 | ~~0~~ 1 | 0 | M[8] | M[12] | 0 | | | |

## 8. Exploration: Improving Cache Penalty

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

- So far, we tried to improve Miss Rate:
  - Larger block size
  - Larger Cache
  - Higher Associativity
- What about **Miss Penalty**?

Thursday, November 30, 2023 2:33 PM

Are you a RAW hazard? Come find out how many cycles you take. Start with 0 cycles.

Q1. Do you have forwarding?
No - Exit with 2 cycles
Yes - Go to Q2

Q2. Is your first instruction lw?
No - Go to Q4 (Forwarding from EX)
Yes - Go to Q3 (Forwarding from MEM)

Q3. Is your second instruction sw?

No - Add one cycle and go to Q4
Yes - Exit (MEM-MEM forward)

Q4. Is your second instruction beq with early branching?
No - Exit (Forwarding to EX/MEM)
Yes - Add 1 cycle and exit (Forwarding to ID)

RAW Dependency (Generic)
If no forwarding, need to wait until WB stage of the first instruction for writing back to the registers(first half)
In the second half, do the ID stage of the second instruction to read registers.
Thus 2 cycle delay.

If there is forwarding, it will occur from EX/MEM register to the EX stage of the second instruction.
Thus no delay.

RAW Dependency (First one is Load)
If there is no forwarding, it's the same as the generic one. Need to wait until WB then read during ID.
Thus 2 cycle delay.

If there is forwarding, it will occur from MEM/WB register to the EX stage of the second instruction.
(Load instruction's data is produced only in MEM stage, unlike EX stage in generic.)
Thus 1 cycle delay.

If there is forwarding and the second instruction is Save Word WITH THE SAME REGISTER AS FIRST ARGUMENT, then there isn't any delay because the info is needed only at MEM of sw
Eg
lw $1, 0($5)
sw $1, 0($6)
Thus 0 cycle delay.

RAW Dependency (Second one is Branch) DO GENERIC FIRST
If there is no early branching, the branch calculation is done at MEM. On top of any existing RAW dependencies, th\ere won't be any extra cycle delays.
0 cycle delay.

If there is early branching, the branch calculation is done at ID. On top of any existing RAW dependencies, there will be an extra +1 delay if there is forwarding, because the data needs to be produced at EX stage, forward to ID.
Thus 1 cycle delay. (If Forwarding)
Thus 0 cycle delay (If no Forwarding)

Control Dependency (First one is Branch)
If there is no branch prediction, the branch calculation is done at MEM or ID depending on setup. On top of any existing early branching or not, there will be a guaranteed 1 cycle delay because you need the calculation to finish before doing the next IF.
Thus 1 cycle delay.

If there is branch prediction, you can run the IF in the background while the branch calculation is happening. If the prediction is wrong, simply flush and IF again (+1 delay like before). If the prediction is correct, no delay as the correct instruction is already there.
Thus 0/1 cycle delay depending on prediction correctness.