CS250

Computer Architecture

Gustavo Rodriguez-Rivera

Computer Science Department

Purdue University

General Information

- Web Page:
  http://www.cs.purdue.edu/homes/cs250
- Office: DS 1139C
- E-mail: grr@purdue.edu
- Textbook:
  - Essentials of Computer Architecture
    Second edition
    D. E. Comer
    Prentice Hall
    0-13-149179-2

Grading

- Grade allocation
  - Midterm: 25%
  - Final: 25%
  - Labs and Homework: 40%
  - Attendance 10%
- Exams also include questions about the projects.
- Attendance will be done with on-line quiz in Brightspace and Partake due during the lecture time.

Course Organization

1. Basics Fundamentals of
   - Digital Logic
   - Data Representation
2. Processors
   - Types of Processors
   - Instruction Sets
   - Assembly Language

Course Organization

3. Memory
   - Types of Memory
   - Physical and Virtual Memory
   - Caching
4. Input/Output
   - Devices and Interfaces
   - Buses
   - Device Drivers

Course organization

5. Advanced Topics

- Parallelism
- Performance Measurement
- Architectural Hierarchy

Approach

- We will cover Computer Architecture
  - From the programmer's point of view.
  - How it influences the programmer’s choices.
- We will not cover
  - Low engineering details
  - VLSI design

II. Fundamentals of Digital Logic

Voltage and Current

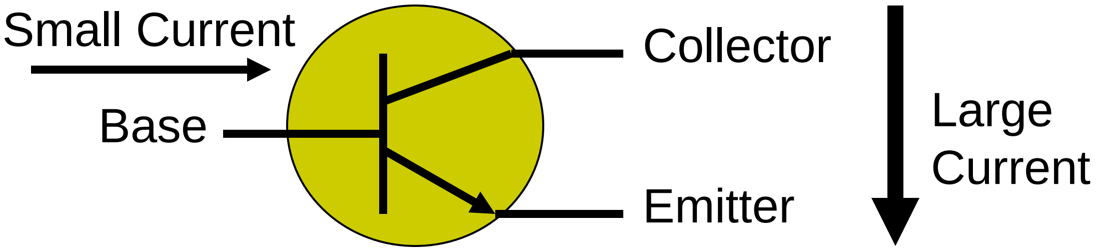
- Voltage
  - Measure of potential Force
  - It is measured in Volts
- Current
  - Measure of electron flow across a wire
  - It is measured in Amps

Voltage

- Voltage is measured with a voltmeter across two points.
- Typical digital circuits work with 5 volts:
  - Ground - 0 volts – represent a “0”
  - Power – 5 volts – represent a “1”

Transistor

- Building block of digital circuits
- Acts like a switch
- A transistor has three connections:
  - Emitter
  - Base
  - Collector



- The current between “Base” and “Emitter” controls the current between “Collector” and “Emitter”.

Boolean Logic

- It gives the formal basis for digital circuits
- It uses three basic functions

AND

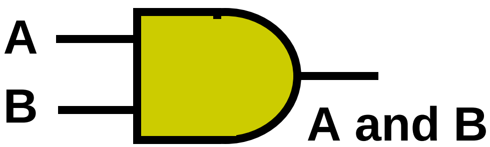
A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

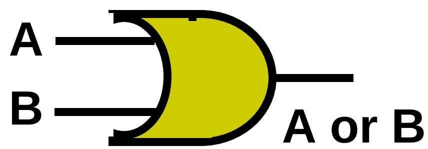
OR

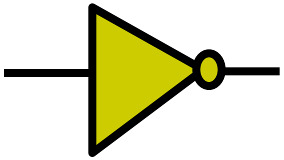
A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

NOT

A	not A
0	1
1	0







Boolean Logic

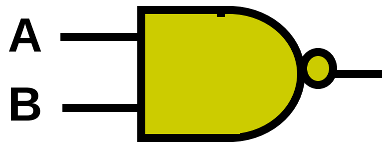
- You will find that Nand and Nor Gates are very popular.
- By using them, there is no need of Not gate

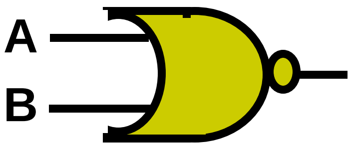
NAND

A	B	A nand B
0	0	1
0	1	1
1	0	1
1	1	0

NOR

A	B	A nor B
0	0	1
0	1	0
1	0	0
1	1	0



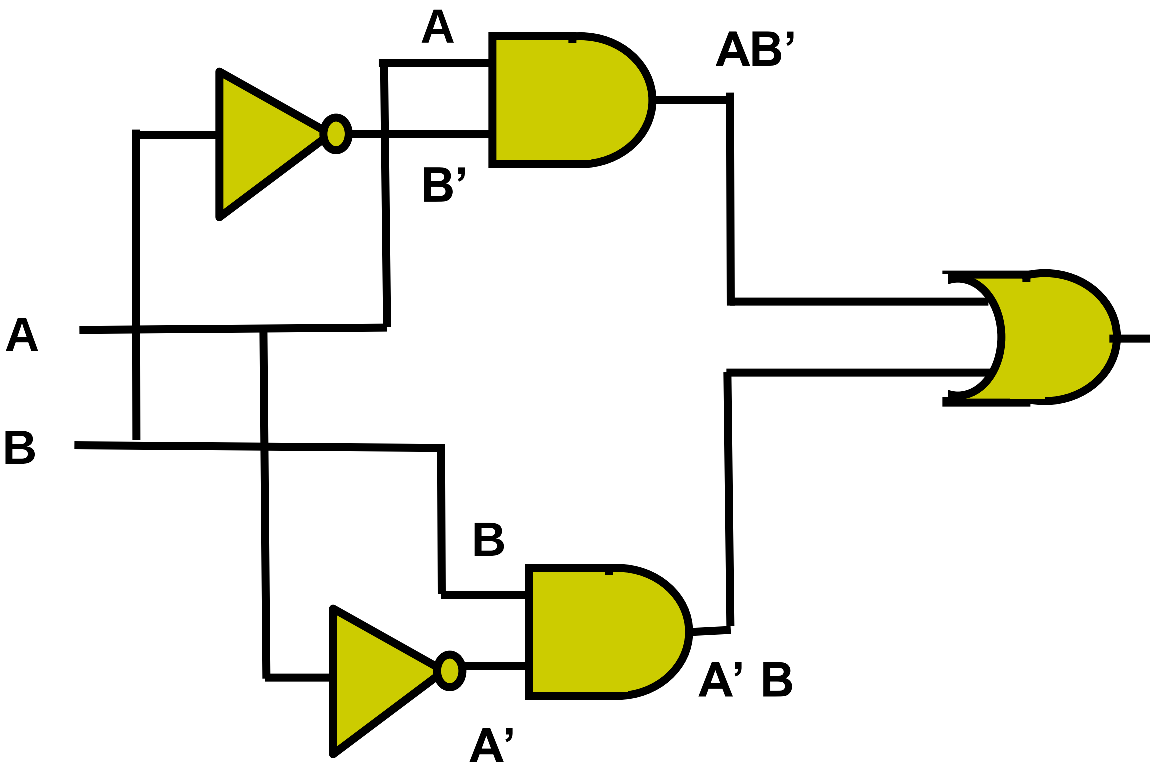


Boolean Logic

- In digital circuits 0 and 1 are represented as
  - 0 = 0 volts
  - 1 = +5 volts
- You can interconnect digital circuits with each other to create complex Boolean expressions.
- (A and B) is represented as AB
- (A or B) is represented as A+B
- (not A) is represented as A’

Boolean Logic

- Example:



Truth Table

A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

Truth Tables to Boolean Expressions

- From a Truth table you can create a boolean expression
- You can represent the boolean function as a
  - Sum of products: Example z=x'y+xy'
  - Product of sums: Example z=(x+y)(x'+y')

Sum of Products

- To create a sum of products from a truth table, take the 1s in z (the output) and use the input for that row to create the product. If the input is x=1 then use x, otherwise if x=0 use x'.

Truth Table

x	y	z
0	0	0
0	1	1
1	0	1
1	1	0

$z = x'y + xy'$

Truth Table

x	y	z
0	0	1
0	1	1
1	0	1
1	1	0

Sum of products:

$z = x'y' + x'y + xy'$

Product of Sums

- To create a product of sums from a truth table, take the 0s in z (the output) and use the variables for that row to create the product. If the input is x=0 then use x, otherwise if x=1 use x'.

Truth Table

x	y	z
0	0	0
0	1	1
1	0	1
1	1	0

$z = (x+y)(x'+y')$

Truth Table

x	y	z
0	0	0
0	1	0
1	0	1
1	1	0

Product of Sums:

z= (x+y)(x+y′)(x′+y′)

- Adding two binary numbers of 1 bit
- A+B=R1 R0
- 0+0=00
- 0+1=01
- 1+0=01
- 1+1=10

Example: Implementing add

- Assume we want to add two numbers where each number will be one bit long.
- The resulting number may be two bits long

		A plus B		
A	**B**	**R1**	**R0**	**In decimal**
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	2

- This can be represented as:
  R0 = A’B+AB’
  R1 = AB

Implementing Add

- To implement an adder for 8 bits or 32 bits, many more gates are required.
- AAAAAAAA
  +BBBBBBBB

- RRRRRRRRR
-
  Truth table:
  AAAAAAAA BBBBBBBB RRRRRRRRR
- 00000000 00000000 000000000
- 00000000 00000001 000000001
- …
- 11111111 11111111 111111111

- 8 bit + 8bits (AsBs)= 16 inputs
- # outputs (Rs) = 9 outputs

- Truth table will have 2^16=65,536 entries

Boolean Algebra

- You can manipulate the boolean expressions like normal algebraic expressions.
- Properties
  - Commutative:
    - AB = BA
    - A+B=B+A
  - Associative:
    - (A+B)+C = A+(B+C)
  - Distributive

  A(B+C) = AB+AC

De Morgan's Law

- Negation of expressions
  - (A+B)' = A'B'
  - (AB)' = A' + B'

Boolean Expression Reduction

- You can simplify Boolean expressions to use fewer gates:
- Example:

  $z = a'b'c + a'b' + ac' + abc'$

  $= a'b'(c+1) + ac'(1+b)$

  $= a'b'+ac'$
- Example:

  $m = x'yz + x'yz' + x'y' + xyz$

  $= x'y(z+z') + x'y' + xyz$

  $= x'y+ x'y' + xyz$

  $= x'(y+y') + xyz$

  $= x' + xyz$

Karnaugh Maps

- To make the simplification of boolean expressions easier, we can use Karnaugh maps.
- A Karnaugh map is a way of expressing truth tables
- Adjacent columns or rows change only by one digit.
- They show when refactoring can be done.

Karnaugh Table example 1

- Given an expression
  r = x'yz'+xyz'+x'y'z+x'yz
- Build a 3 variable Karnaugh map
- Find the groups of 2, 4 or 8 1's that are adjacent.
- Make sure ***all*** 1s are covered by the largest groups with the minimum number of groups. They may overlap.
- Build expression from groups.

r = yz' + x'z

Karnaugh Map for r

xy z	00	01	11	10
0	0	1	1	0
1	1	1	0	0

xy z	00 (x'y')	01 (x'y)	11 (xy)	10 (xy')
0(z')	0	1x'yz'	1xyz'	0
1(z)	1	1	0	0

r = yz' + x'z

Karnaugh Table example 2

- Given an expression

  r = x'yz'k'+x'yz'k+xyz'k'+xyz'k+ xyzk+xyzk'+x'y'zk'+xy'zk'
- Build a 4 variable Karnaugh map
- Find the largest groups of 2, 4, 8 or 16 1's that are adjacent.
- Make sure all 1s are covered by the groups.
- Only use the minimum number of groups that cover the 1s
- Overlapping is good since it gives a smaller expression.
- Build expression from groups.

r = yz' + xy + y'zk'

Karnaugh Map for r

xy/ zk	00 (x'y')	01 (x'y)	11 (xy)	10 (xy')
00 (z'k')	0	1	1	0
01 (z'k)	0	1	1	0
11 (zk)	0	0	1	0
10 (zk')	1	0	1	1

r = yz' + xy + y'zk'

Using only NAND Gates

- Very often you build the circuits using only NAND gates.
- To convert a sum of products to only NAND gates negate the function twice and reduce
- Example:

  z = x XOR y = xy'+x'y

Now if you negate twice the right side and applying De Morgans law.

z = ((xy'+x'y)')' = ((xy')'(x'y)')' = (x NAND y') NAND (x' NAND y)

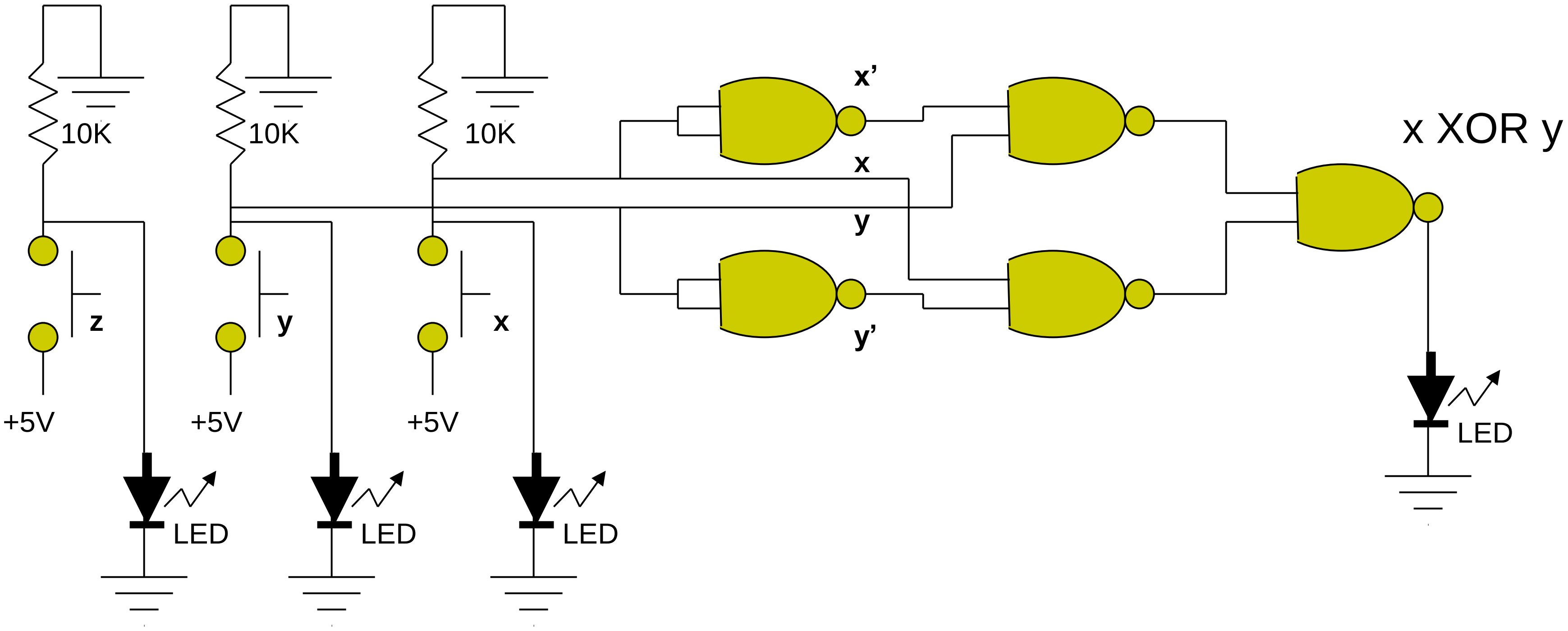
Also, since x'= (x x)' = x NAND x and y' = y NAND y then we have:

z = (x NAND (y NAND y)) NAND ((x NAND x) NAND y)

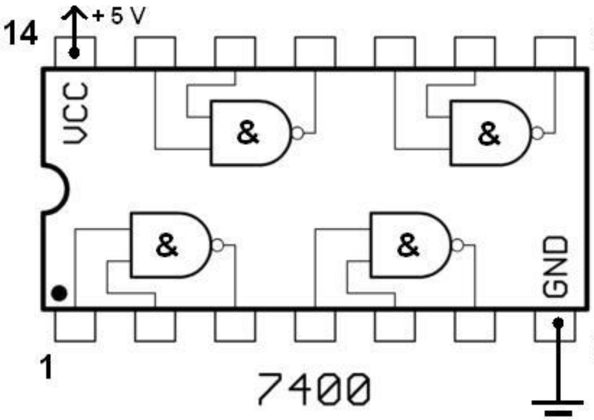
(Alternatively x'=(x and 1)'=x NAND 1

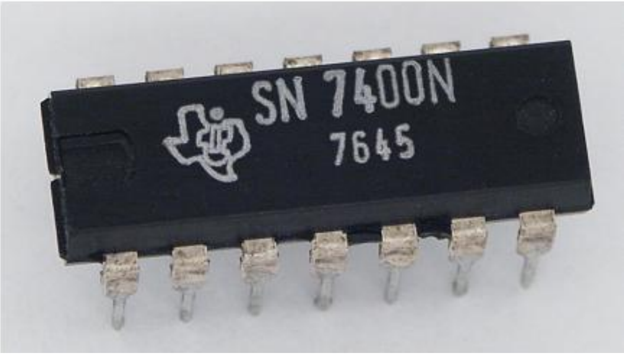
XOR Using only NAND gates

x XOR y = (x NAND (y NAND y)) NAND ((x NAND x) NAND y)



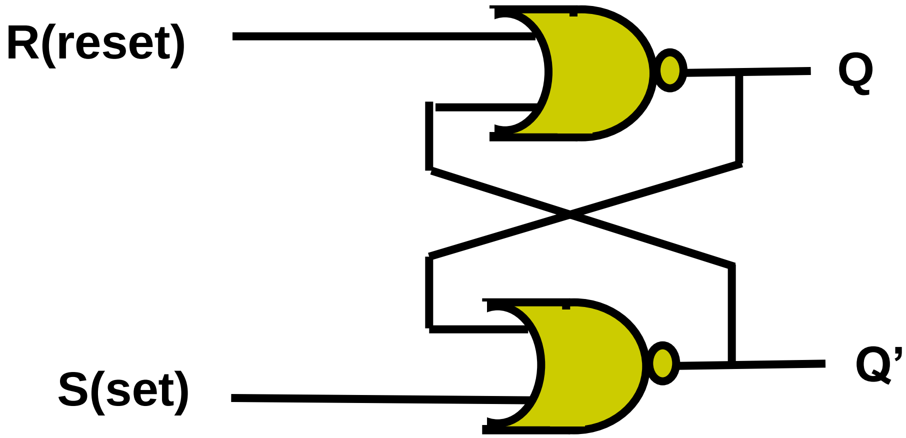
Examples of Gates on 7400-Series Chips





RS Flip Flops

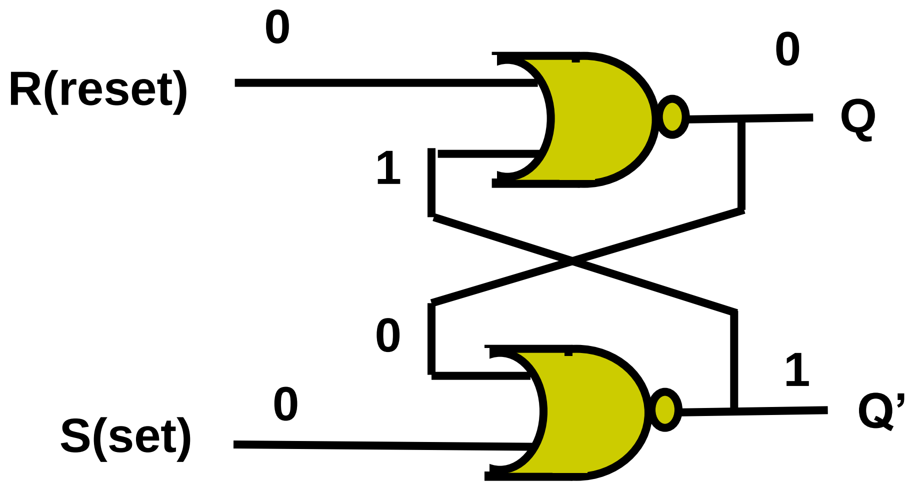
- Basic unit of memory

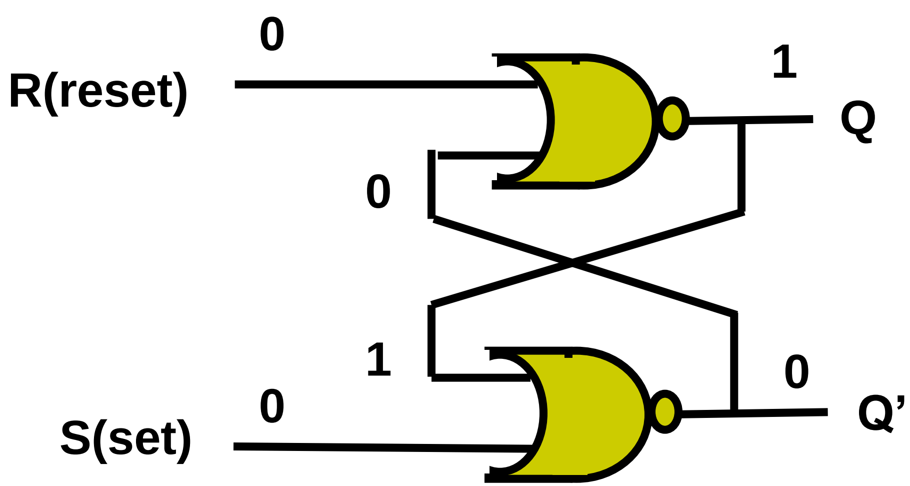


Truth Table

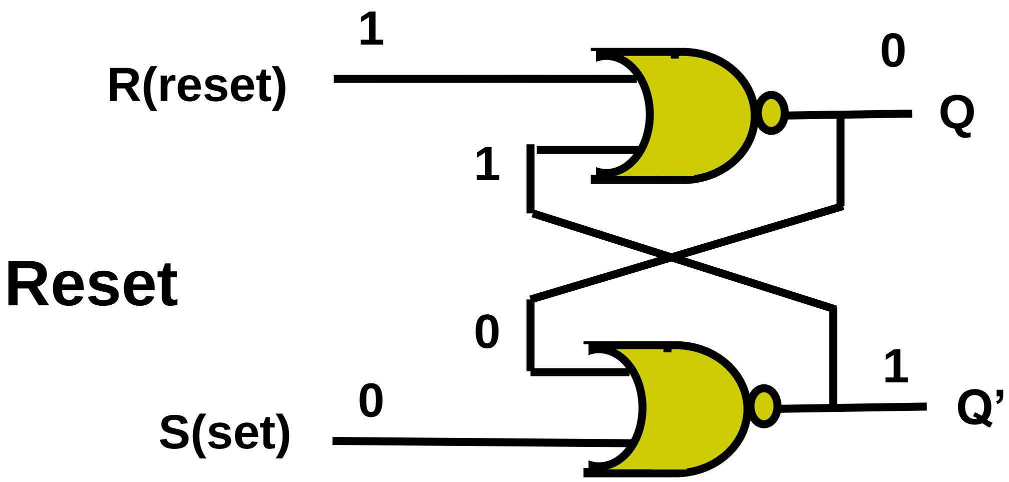
S	R	Q	Q'
0	0	Keep previous value	
0	1	0	1
1	0	1	0
1	1	Not allowed	

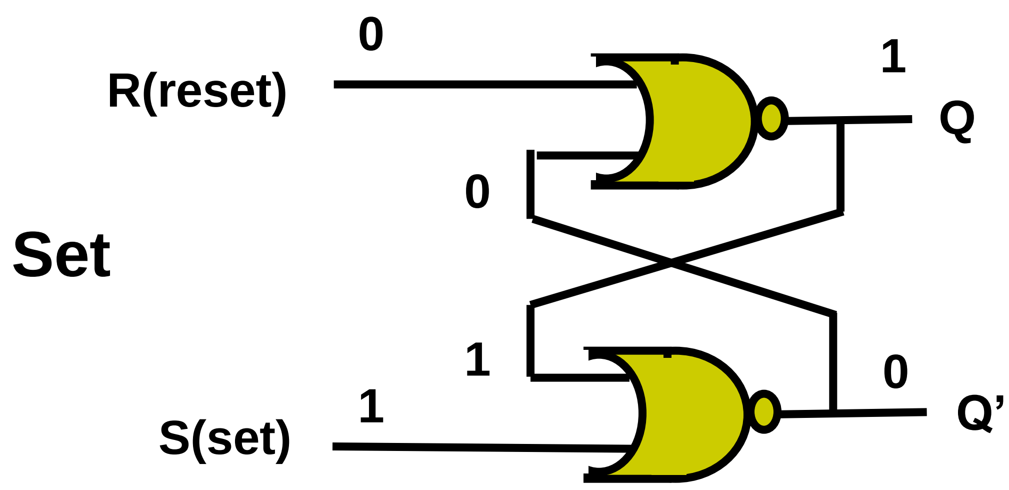
RS Flip Flops. Keep Current value





RS Flip Flops. Reset and Set

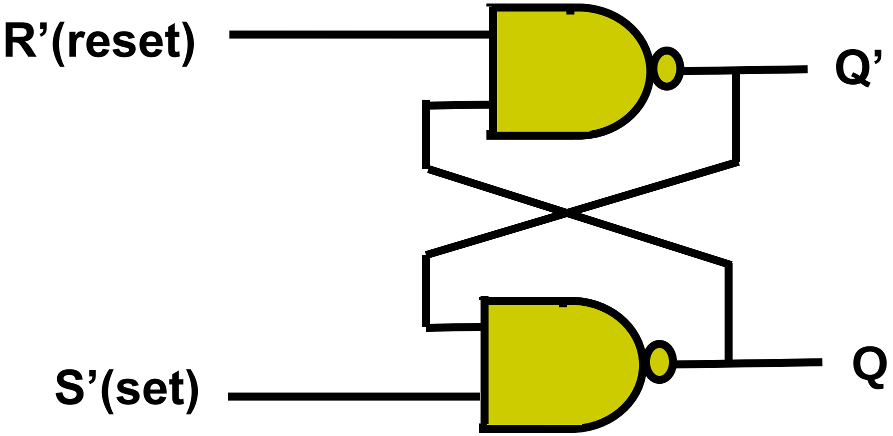




The Input R=1 and S=1 is not allowed.

R’S’ Flip Flops

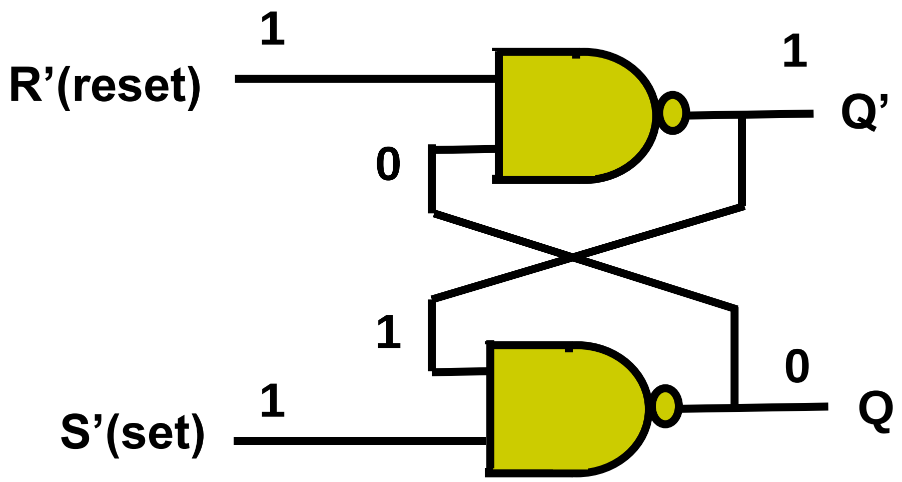
- Flip-Flops that use NANDs
- Inverted input and outputs

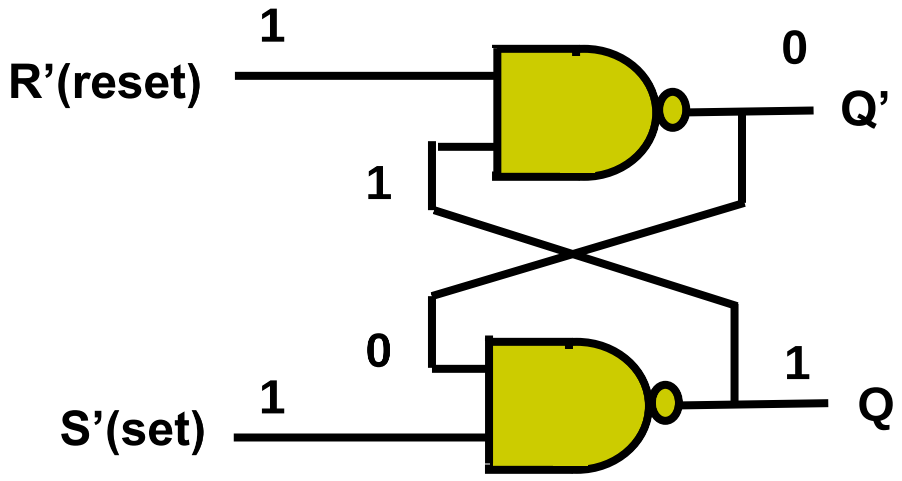


Truth Table

S′	R′	Q′	Q
0	0	Not allowed	
0	1	0	1
1	0	1	0
1	1	Keep previous value	

R'S' Keep Current Value

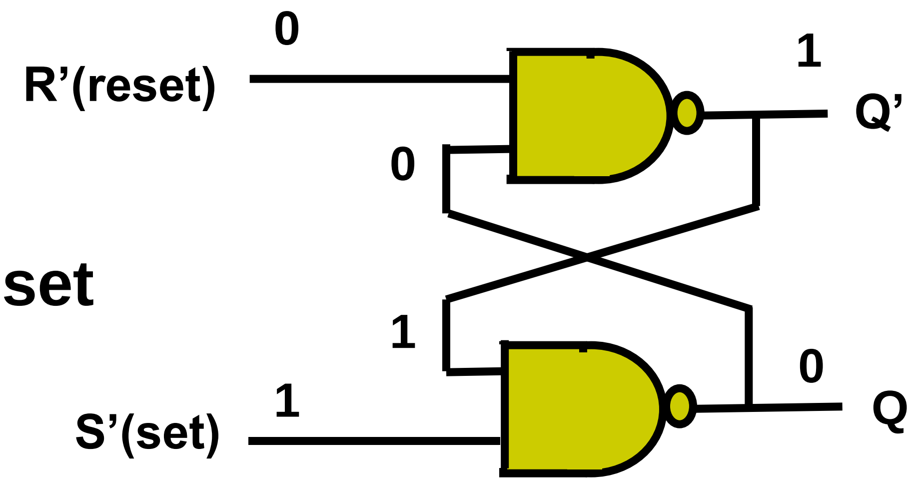




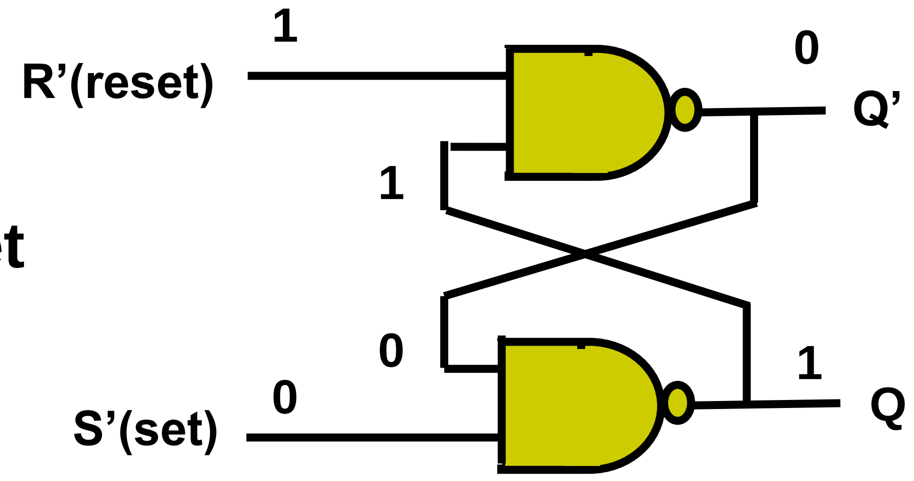
The Input R'=0 and S'=0 is not allowed.

R'S' Flip Flops. Use NANDs

Reset



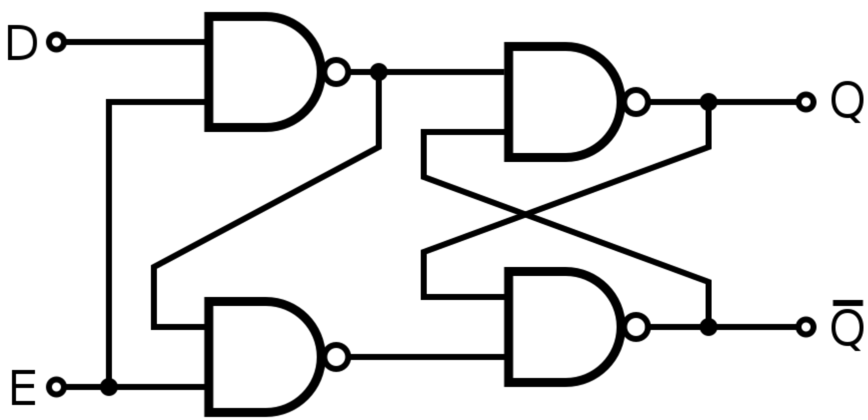
Set



The Input R=1 and S=1 is not allowed.

D-Latch

- Modification of R’S’ Flip-Flop to prevent not allowed input
- Adds Enable input to keep or set value from input D



Truth Table

E	D	Q’	Q
0	0	Keep previous value	
0	1	Keep previous value	
1	0	1	0
1	1	0	1

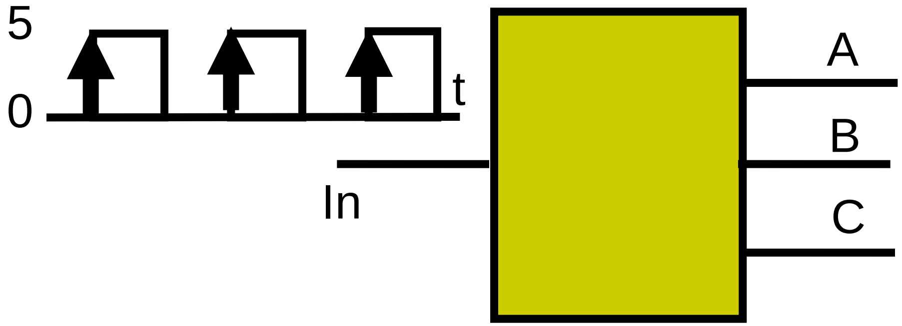
Lab 3 – Sequential Circuits

- Step 1 – Implement R'S' Flip Flop.
  - Add push buttons for input. Use Active Low configuration from Lab 1
  - Add Output with LEDs
- Step 2 – Implement D-Latch (optional)

Binary Counter

- Counts pulses (transitions from 0 to 1)
- Output is a binary number
- Contains a terminal to reset ouput to 0

Binary Counter (3 bits)

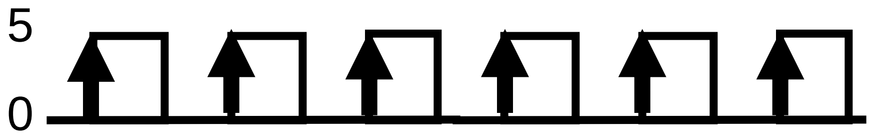


Truth Table

In	A	B	C
0	0	0	0
1	0	0	1
0	0	0	1
1	0	1	0
0	0	1	0
1	0	1	1
0	0	1	1
.	.	.	.

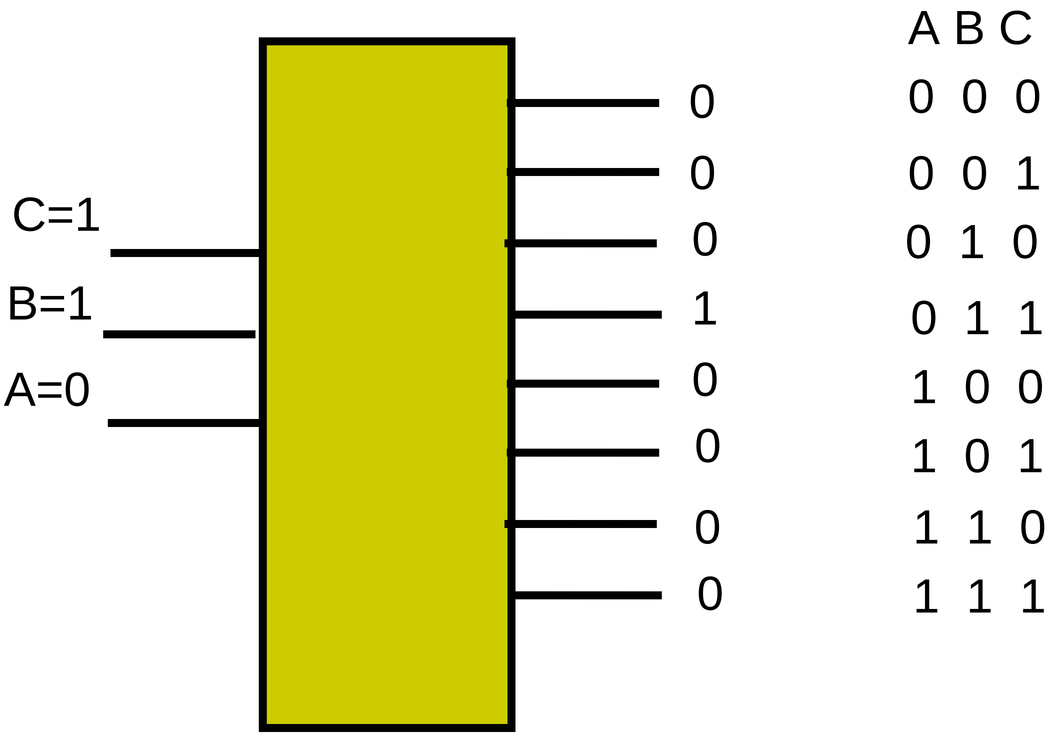
Clock

- It is an electronic circuit that produces a sequences of 0 1 0 1 0 1
- The frequency is measured in hertz (Hz).
- It is used to synchronize operations across gates in active circuits.

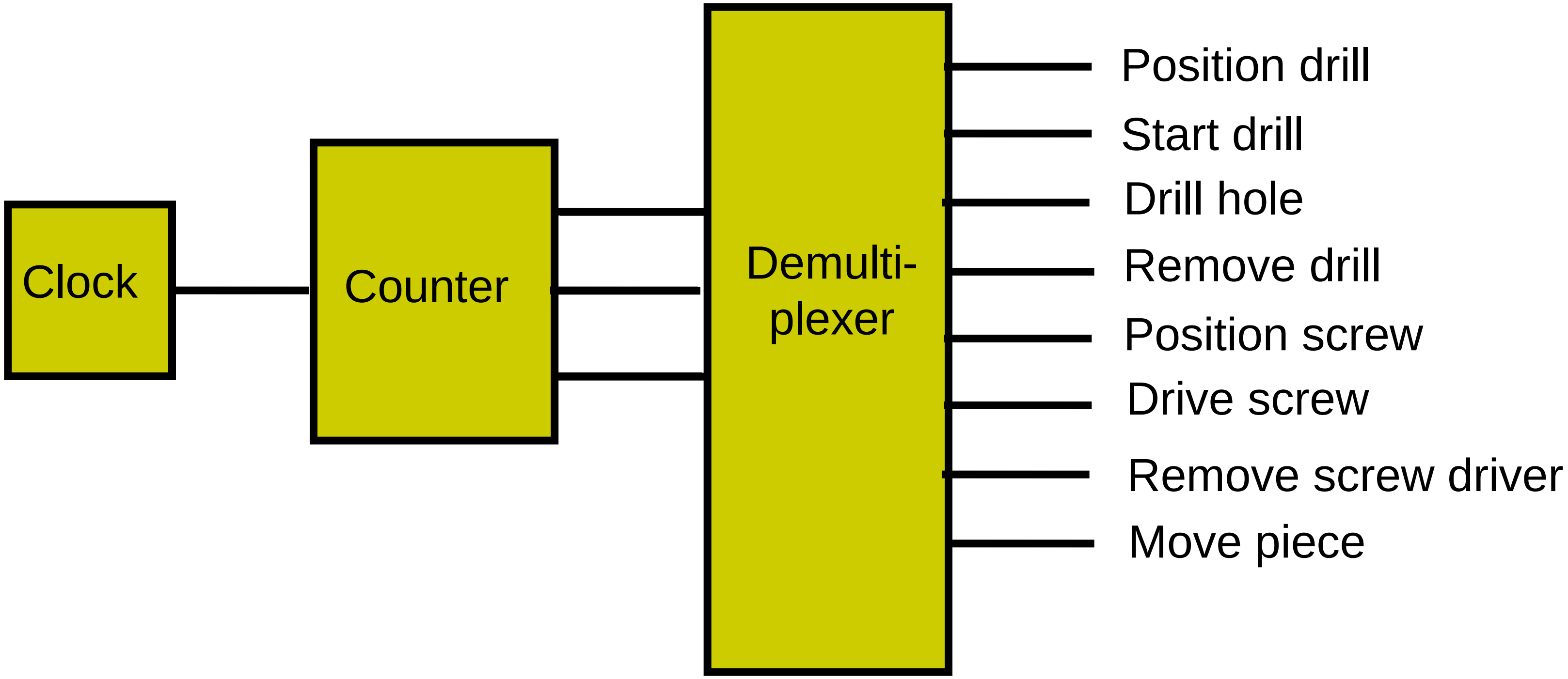


Demultiplexor

- It is a circuit used to select one output



Example of Circuit to Execute a Sequence of Steps



Unused Gates

- Since a chip may contain multiple gates, it is possible to use some of the spare gates to do other operations instead of adding a new chip.
- Example:
  - 1 nand x = not x

Classification of Technologies

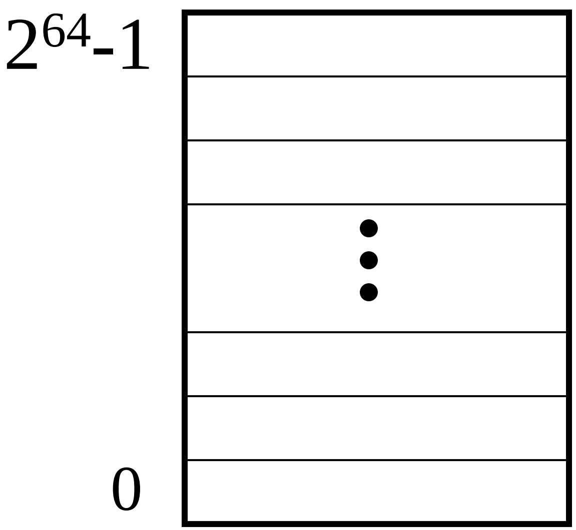
- Small Scale Integration (SSI)
  - Basic Boolean Gates
- Medium Scale Integration (MSI)
  - Intermediate logic such as demultiplexers and counters
- Large Scale Integration (LSI)
  - Small embedded processors
- Very Large Integration (VLSI)
  - Complex processors

  - http://data.cs.purdue.edu:3000

III. Data and Program Representation

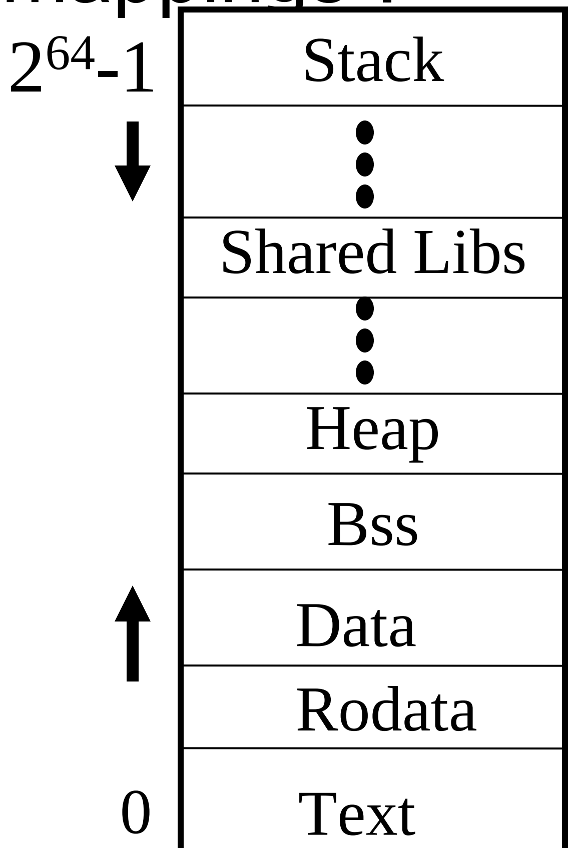
Memory of a Program

- A program sees memory as an array of bytes that goes from address 0 to 2^{64}-1
- That is assuming a 64-bit architecture.
- Most computers have 8GB-24GB RAM
- Not all the address space is used.
- 2^10=IKB=1024
- 2^20=1MB=1024*1024
- 2^30=1GB=1024*1024*1024
- 2^40=1TB=1024*1024*1024*1024
- 2^64=2^24*2^40=2^24TB~=16M TB



Memory Sections

- The memory is organized into sections called "memory mappings".



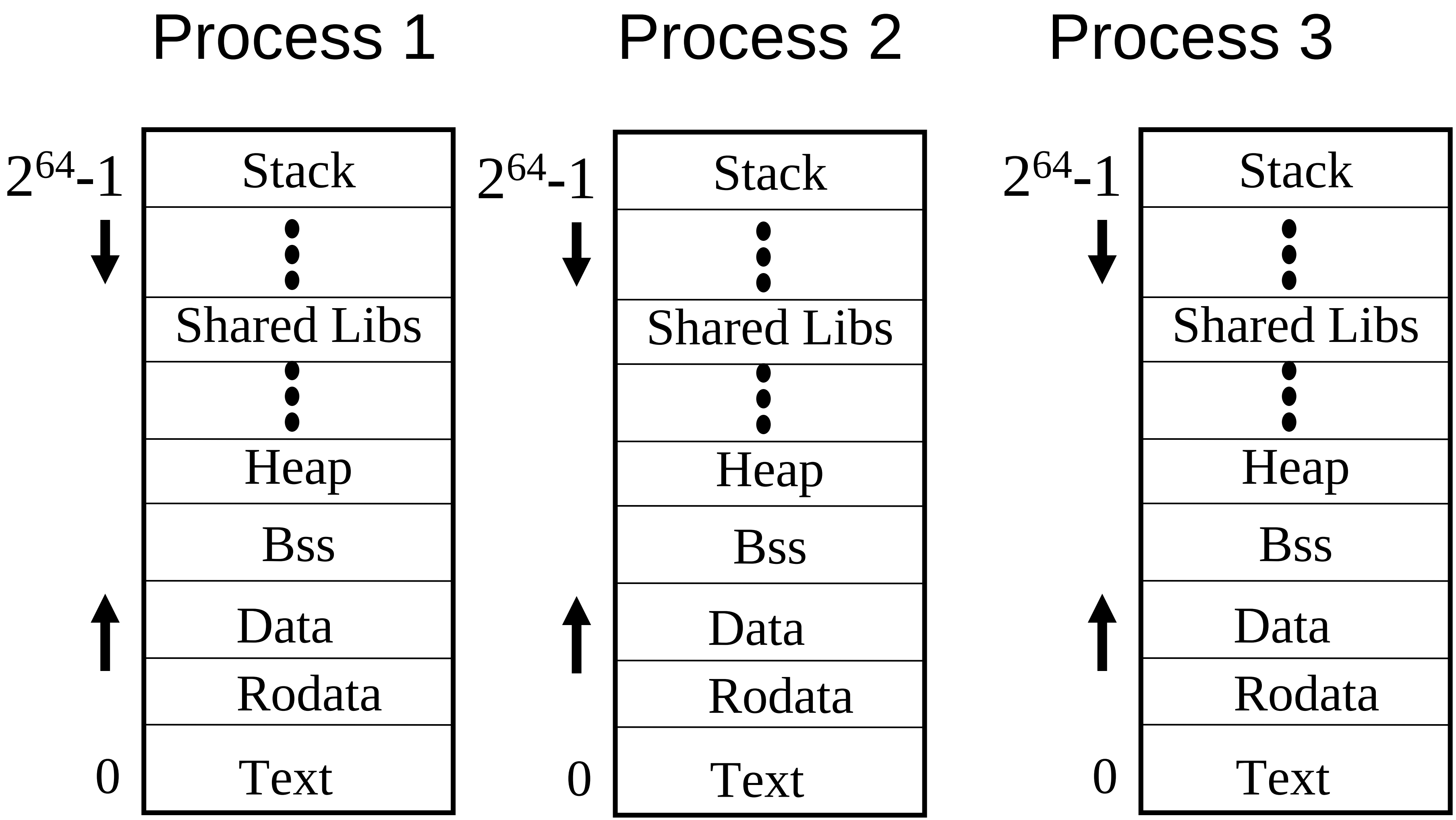
Memory Sections

- Each section has different permissions: read/write/execute or a combination of them.
- Text- Instructions that the program runs
- Rodata – Read Only Data. String constants
- Data – Initialized global variables.
- Bss – Uninitialized global variables. They are initialized to zeroes.
- Heap – Memory returned when calling malloc/new. It grows upwards.
- Stack – It stores local variables and return addresses. It grows downwards.

Memory Sections

- Dynamic libraries (Shared Libraries) – They are libraries shared with other processes.
- Each dynamic library has its own text, data, and bss.
- Each program (process) has its own view of the memory that is independent of each other.
- This view is called the “Address Space” of the program.
- If a process modifies a byte in its own address space, it will not modify the address space of another process.

Processes have their own memory address space



Example

```
Program hello.c
int a = 5;    // Stored in data section
int b[20];    // Stored in bss
int main() { // Stored in text
  int x;      // Stored in stack
  int *p;
  p =(int*)
     malloc(sizeof(int)); //In heap
  *p = 5;
  printf("Hello cs250\n");
}
```

Memory Gaps

- Between each memory section there may be gaps that do not have any memory mapping.
- If the program tries to access a memory gap, the OS will send a SEGV signal that by default kills the program and dumps a core file.
- The core file contains the value of the variables global and local at the time of the SEGV.
- The core file can be used for “post mortem” debugging.

  gdb program-name core

  gdb> where

What is a program?

- A program is a file in a special format that contains all the necessary information to load an application into memory and make it run.
- A program file includes:
  - machine instructions
  - initialized data
  - List of library dependencies
  - List of memory sections that the program will use
  - List of undefined values in the executable that will be known when the program is loaded into memory.

Executable File Formats

- There are different executable file formats
  - ELF – Executable Link File

    It is used in most UNIX systems (Solaris, Linux)
  - COFF – Common Object File Format

    It is used in Windows systems
  - a.out – Used in BSD (Berkeley Standard Distribution) and early UNIX

    It was very restrictive. It is not used anymore.
- Note: BSD UNIX and AT&T UNIX are the predecessors of the modern UNIX flavors like Solaris and Linux.

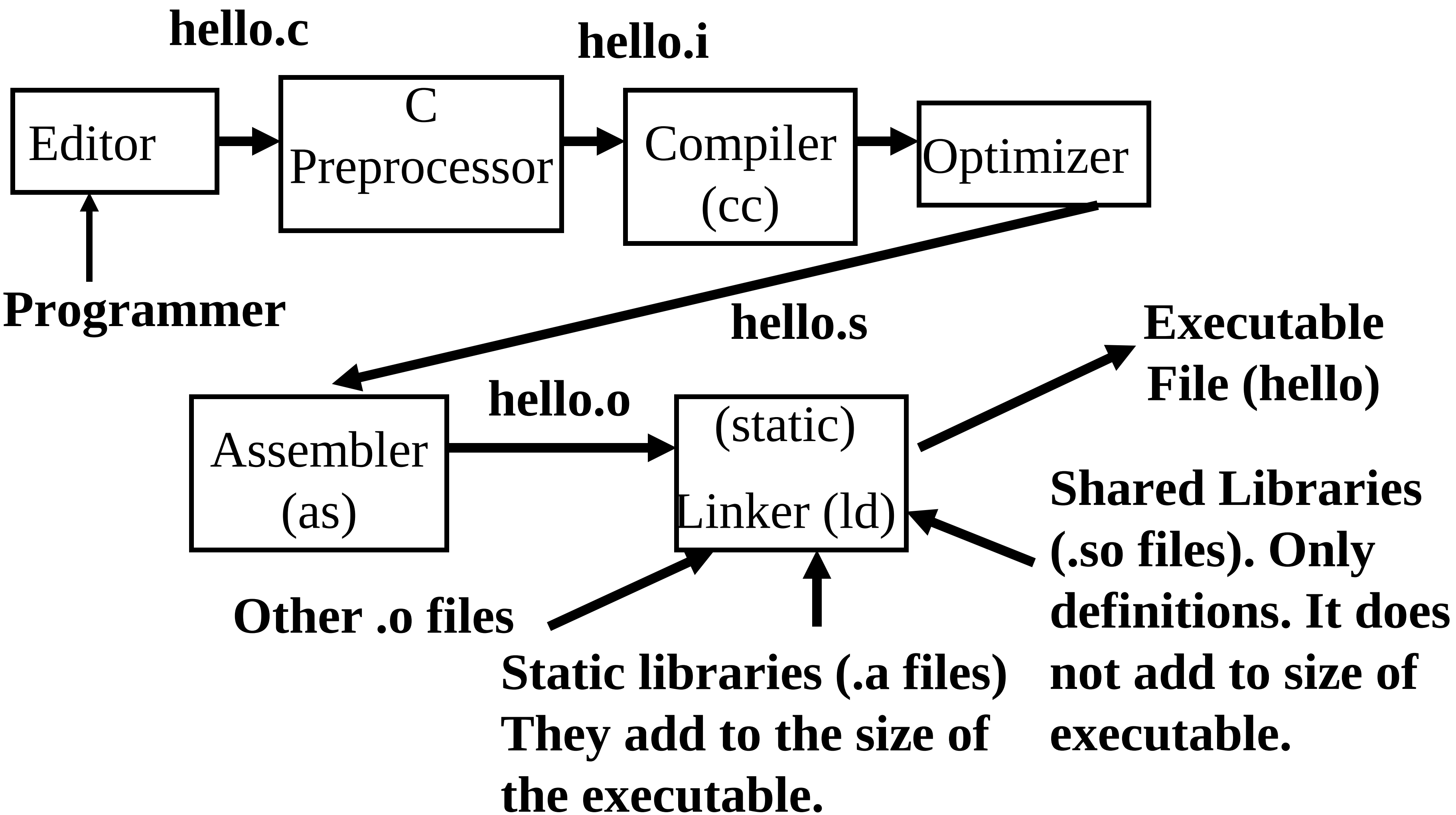
Building a Program

- The programmer writes a program hello.c
- The ***preprocessor*** expands #define, #include, #ifdef etc preprocessor statements and generates a hello.i file.
- The ***compiler*** compiles hello.i, optimizes it and generates an assembly instruction listing hello.s
- The ***assembler*** (as) assembles hello.s and generates an object file hello.o
- The compiler (cc or gcc) by default hides all these intermediate steps. You can use compiler options to run each step independently.

Building a program

- The linker puts together all object files as well as the object files in static libraries.
- The linker also takes the definitions in shared libraries and verifies that the symbols (functions and variables) needed by the program are completely satisfied.
- If there is symbol that is not defined in either the executable or shared libraries, the linker will give an error.
- Static libraries (.a files) are added to the executable. shared libraries (.so files) are not added to the executable file.

Building a Program



Original file hello.c

```
#include <stdio.h>
#define PI 3.14
int main()
{
    printf("Hello\n");
    printf("PI=%lf\n",PI);


}
```

After preprocessor

```
gcc -E hello.c > hello.i
    (-E stops compiler after running preprocessor)
  hello.i:
      /* Expanded /usr/include/stdio.h */
    typedef void *__va_list;
    typedef struct __FILE  __FILE;
    typedef int     ssize_t;
    struct FILE {…};
    extern int fprintf(FILE *, const char *, ...);
    extern int fscanf(FILE *, const char *, ...);
    extern int printf(const char *, ...);
    /* and more */
    main()
    {
       printf("Hello\n");
       printf("PI=%lf\n",3.14);
    }
```

After assembler

gcc -S hello.c   (-S stops compiler after assembling)

hello.s:

```
            .align 8
.LLC0:  .asciz  "Hello\n"
.section        ".text"
        .align 4
        .global main
        .type    main,#function
        .proc   04
main:   save    %sp, -112, %sp
        sethi   %hi(.LLC0), %o1
        or      %o1, %lo(.LLC0), %o0
        call    printf, 0
        nop
.LL2:   ret
        restore
.
```

After compiling

- "gcc -c hello.c" generates hello.o
- hello.o has undefined symbols, like the ***printf*** function call that we don't know where it is placed.
- The main function already has a value relative to the object file hello.o

csh> nm hello.o

```
hello.o:
[Index]    Value        Size        Type  Bind  Other Shndx   Name
[1]       |0x00000000|0x00000000|FILE |LOCL |0     |ABS    |hello.c
[2]       |0x00000000|0x00000000|NOTY |LOCL |0     |2      |gcc2_compiled
[3]       |0x00000000|0x00000000|SECT |LOCL |0     |2      |
[4]       |0x00000000|0x00000000|SECT |LOCL |0     |3      |
[5]       |0x00000000|0x00000000|NOTY |GLOB |0     |UNDEF  |printf
[6]       |0x00000000|0x0000001c|FUNC |GLOB |0     |2      |main
```

After linking

- **“gcc –o hello hello.c”** generates the hello executable
- Printf does not have a value yet until the program is loaded

```
csh> nm hello
[Index]   Value      Size      Type  Bind  Other Shndx   Name
[29]     |0x00010000|0x00000000|OBJT |LOCL |0    |1      |_START_
[65]     |0x0001042c|0x00000074|FUNC |GLOB |0    |9      |_start
[43]     |0x00010564|0x00000000|FUNC |LOCL |0    |9      |fini_dummy
[60]     |0x000105c4|0x0000001c|FUNC |GLOB |0    |9      |main
[71]     |0x000206d8|0x00000000|FUNC |GLOB |0    |UNDEF  |atexit
[72]     |0x000206f0|0x00000000|FUNC |GLOB |0    |UNDEF  |_exit
[67]     |0x00020714|0x00000000|FUNC |GLOB |0    |UNDEF  |printf
```

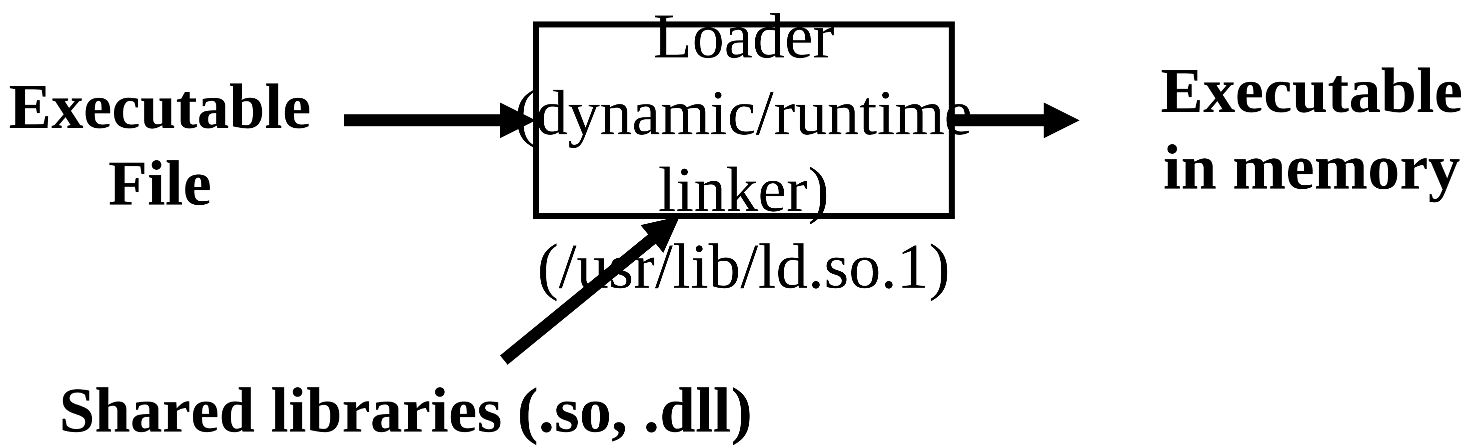
Loading a Program

- The loader is a program that is used to run an executable file in a process.
- Before the program starts running, the loader allocates space for all the sections of the executable file (text, data, bss etc)
- It loads into memory the executable and shared libraries (if not loaded yet)

Loading a Program

- It also writes (resolves) any values in the executable to point to the functions/variables in the shared libraries.(E.g. calls to printf in hello.c)
- Once memory image is ready, the loader jumps to the _***start*** entry point that calls init() of all libraries and initializes static constructors. Then it calls ***main()*** and the program begins.
- _***start*** also calls ***exit()*** when ***main()*** returns.
- The loader is also called “runtime linker” or “dynamic linker”.

Loading a Program



Static and Shared Libraries

- Shared libraries are shared across different processes.
- There is only one instance of each shared library for the entire system.
- Static libraries are not shared.
- There is an instance of a static library for each process.

Static and Dynamic

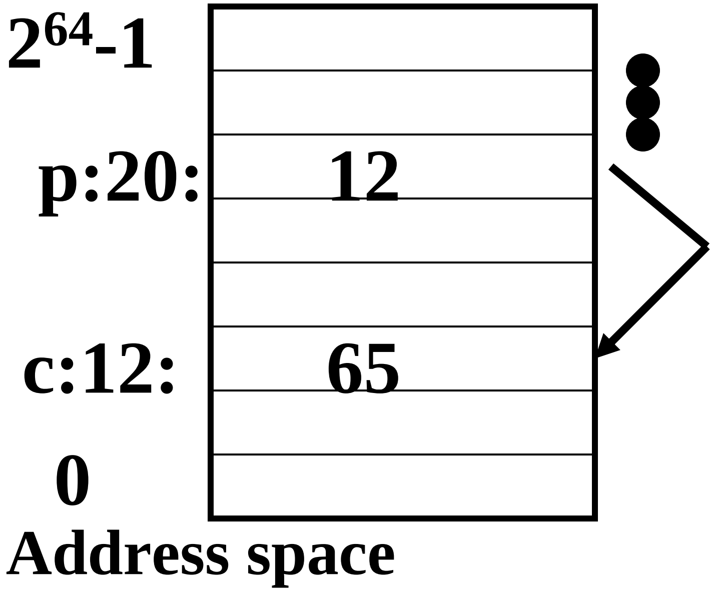
- Static – Events that happen during program building.
  - Example: Static linker,
    Static type checking.
- Dynamic – Events that happen while program is running.
  - Also called “Runtime”.
  - Example: Dynamic linker or
    Runtime linker,
    Dynamic Type checking

- http://data.cs.purdue.edu:3000/

Memory and Pointers

- A pointer is a variable that contains an address in memory.
- In a 64 bit architectures, the size of a pointer is 8 bytes independent on the type of the pointer.

```
char c;
c = ‘A’;  //ascii 65

char * p;

p =  &c;

printf(“*p=%c\n”, *p);//prints A

printf(“*p=%d\n”, *p);//prints 65
```



Ways to get a pointer value

1. Assign a numerical value into a pointer

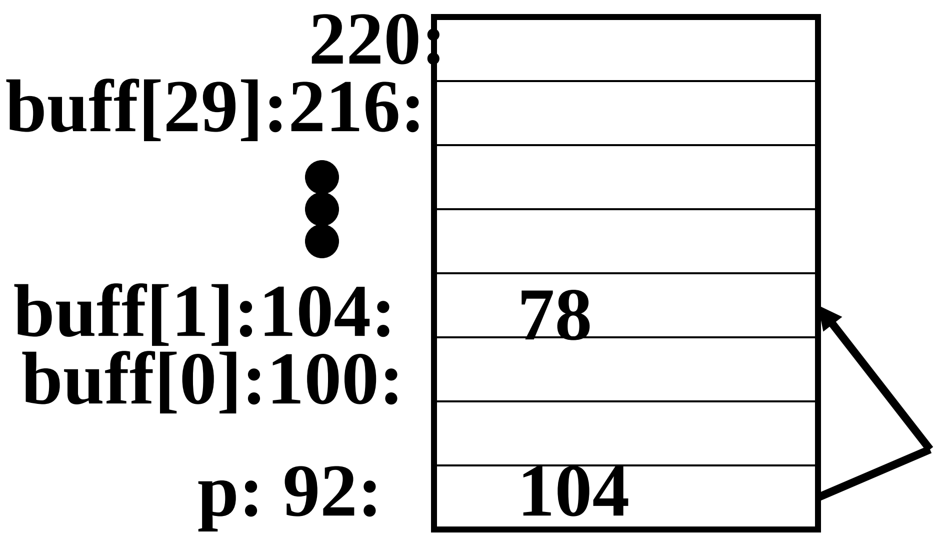
```
char * p;
p = (char *) 0x1800;
*p = 5; // Store a 5 in location 0x1800;
```

Note: Assigning a numerical value to a pointer isn't recommended and only left to programmers of OS, kernels, or device drivers

Ways to get a pointer value

2. Get memory address from another variable:

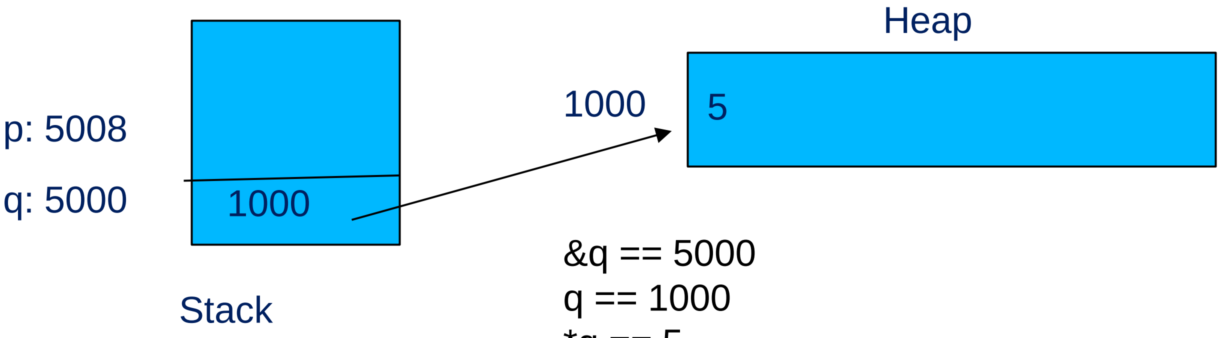
```
int *p;
int buff[ 30];
p = &buff[1];
*p =78;
```



Ways to get a pointer value

3. Allocate memory from the heap

```
int *p;
p = new int;
int *q;
q = (int*)malloc(sizeof(int))
*q = 5
```



&q == 5000
q == 1000
*q == 5

Ways to get a pointer value

- 4. You can pass a pointer as a parameter to a function if the function will modify the content of the parameters

```
void swap (int *a, int *b){
        int temp;
        temp=*a;
        *a=*b;
        *b=temp;
}
In main:
int x = 5;
int y = 6;
swap(&x, &y)
```

Common Problems with Pointers

- When using pointers make sure the pointer is pointing to valid memory before assigning or getting any value from the location
- String functions like strcpy, strcat, strstr etc, do not allocate memory for you:
  char *s;
  strcpy(s, "hello"); --> SEGV(uninitialized pointer)
- The only string function that allocates memory is strdup (it calls malloc of the length of the string plus one and copies it)

Printing Pointers

- **It is useful to print pointers for debugging**

```
char*i;
char buff[10];
printf("ptr=%ld\n",(unsigned long)&buff[5])
```

Or In hexadecimal

```
printf("ptr=0x%lx\n",(unsigned long) &buff[5])
```

Or

```
printf("ptr=%p\n", &buff[5])
```

Instead of using printf, I recommend to use `fprintf(stderr, …)` since stderr is unbuffered and it is guaranteed to be printed on the screen.

`sizeof()` operator in Pointers

- The size of a pointer is always 8 bytes in a 64 bit architecture independent of the type of the pointer:

```
sizeof(int)==4 bytes
sizeof(char)==1 byte
sizeof(long)==8 bytes
sizeof(int*)==8 bytes
sizeof(char*)==8 bytes
sizeof(long*)== 8 bytes
sizeof(char**)== 8 bytes
```

String Operations

- A string is represented in memory as a sequence of characters in ASCII terminated by a ‘\0’ (ASCII Null).

    char a[6];

    strcpy(a,”Hello”);

- Assuming that “a” is at location 1000:

H(72)	E(101)	L(108)	L(108)	O(111)	\0(0)	
1000	1001	1002	1003	1004	1005	

- The string will use one byte more than the length of the string.

String Operations

- The C library (libc) provides simple string functions to manipulate strings such as:
  - char * strcpy(char *dest, char *src)
    - Copies string from “src” to “dest” including char at the end. It assumes that there is enough memory already in “dest”. It does not allocate memory. It returns “dest”.
  - char * strcat(char *dest, char *src)
    - Appends string “src” at the end ofdest. It assumes that there is enough memory already in “dest”. It returns “dest”.
  - char * strstr(char * hay, char * needle)
    - Returns a pointer of the first occurrence of the string “needle” in the string “hay”.

String Operations

- In general the string functions will not allocate memory.
- You have to allocate enough memory before using them.
- The only string function that allocates memory is strdup(char * s) that allocates memory using “malloc” and returns a copy of the string passed in “s”.
- Strdup will allocate strlen(s)+1 characters using malloc and copy s into this memory.
- Also remember if char *s is the string, then sizeof(s)==8bytes in a 64bit architecture and not the length of the string.
- To get the length of the string use strlen(s).

Using Pointers to Optimize Execution

- Assume the following function that adds the sum of integers in an array using array indexing.

```
int sum(int * array, int n)
{
  int s=0;
  for(int i=0; i<n; i++)
  {
    s+=array[i]; // array[i] is equivalent to
         //*(int*)((char*)array+i*sizeof(int))
  }
  return s;
}
```

Using Pointers to Optimize Execution

- Now the equivalent code using pointers

```
int sum(int* array, int n)
{
   int s=0;
   int *p=&array[0];
   int *pend=&array[n];
   while (p < pend)
   {
     s+=*p;
     p++;
   }
   return s;
}
```

Using Pointers to Optimize Execution

- When you increment a pointer to integer it will be incremented by 4 units because sizeof(int)==4.
- Using pointers is more efficient because no indexing is required and indexing require multiplication.
- Note: An optimizer may substitute the multiplication by a “<<“ operator if the size is a power of two. However, the array entries may not be a power of 2 and integer multiplication may be needed.

Array Operator Equivalence

- We have the following equivalences:

```
int a[20];
a[i]          - is equivalent to
*(a+i)        - is equivalent to
*(&a[0]+i)    - is equivalent to
*((int*)((char*)&a[0]+i*sizeof(int)))
```

- You may substitute array indexing `a[i]` by `*((int*)((char*)&a[0]+i*sizeof(int)))` and it will work!
- ***C was designed to be machine independent assembler***

2D Array. 1st Implementation

- 1st approach

Normal 2D array.

```
int a[4][3];
```

```
a[i][j] ==
*(int*)((char*)a +
i*3*sizeof(int) +
j*sizeof(int))
```

	a[3][2]:144:	
	a[3][1]:140:	
	a[3][0]:136:	
	a[2][2]:132:	
	a[2][1]:128:	
	a[2][0]:124:	
	a[1][2]:120:	
	a[1][1]:116:	
	a[1][0]:112:	
	a[0][2]:108:	
	a[0][1]:104:	
a:	a[0][0]:100:	

2D Array 2nd Implementation (Jagged Arrays)

- 2nd approach

Array of pointers to rows

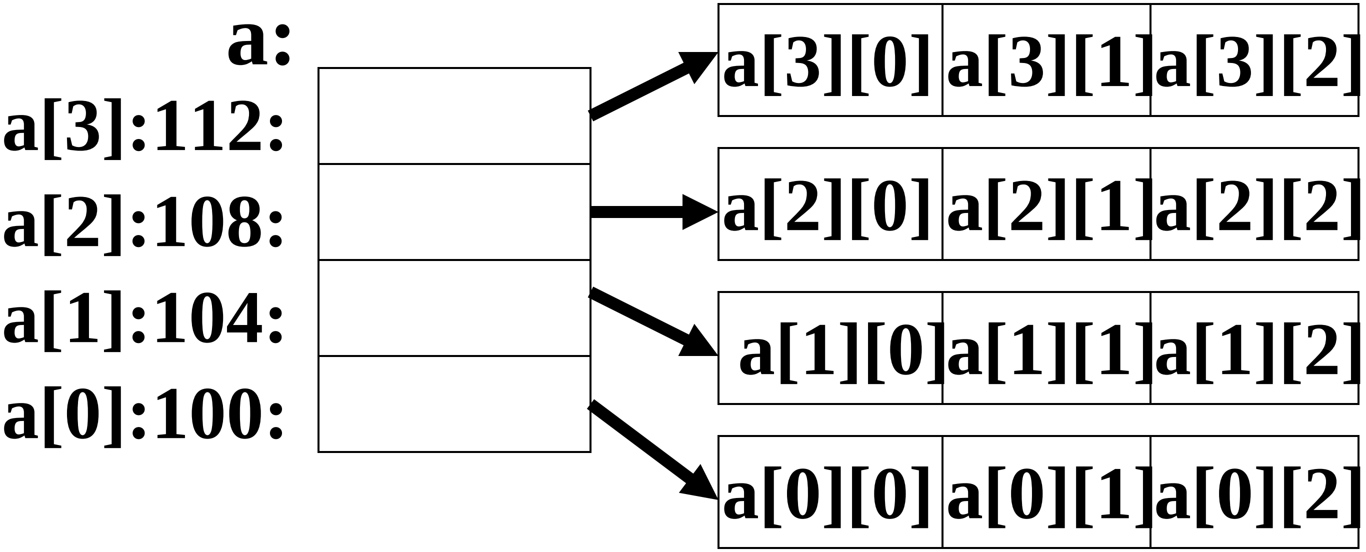
```
int*(a[4]);
for(int i=0; i<4; i++){
      a[i]=(int*)malloc(sizeof(int)*3);
      assert(a[i]!=NULL);
}
```

```
Note:
  int *(a[4]) == a is an array of 4 elements of
  pointer to int. It is the same as int *a[4].
  int (*a)[4] == a is a pointer to an array of 4
  elements of type int.
```

2D Array 2nd Implementation

- 2nd approach

Array of pointers to rows (cont)




```
int*(a[4]);
```

```
a[3][2]=5
```

2D Array 3rd Implementation

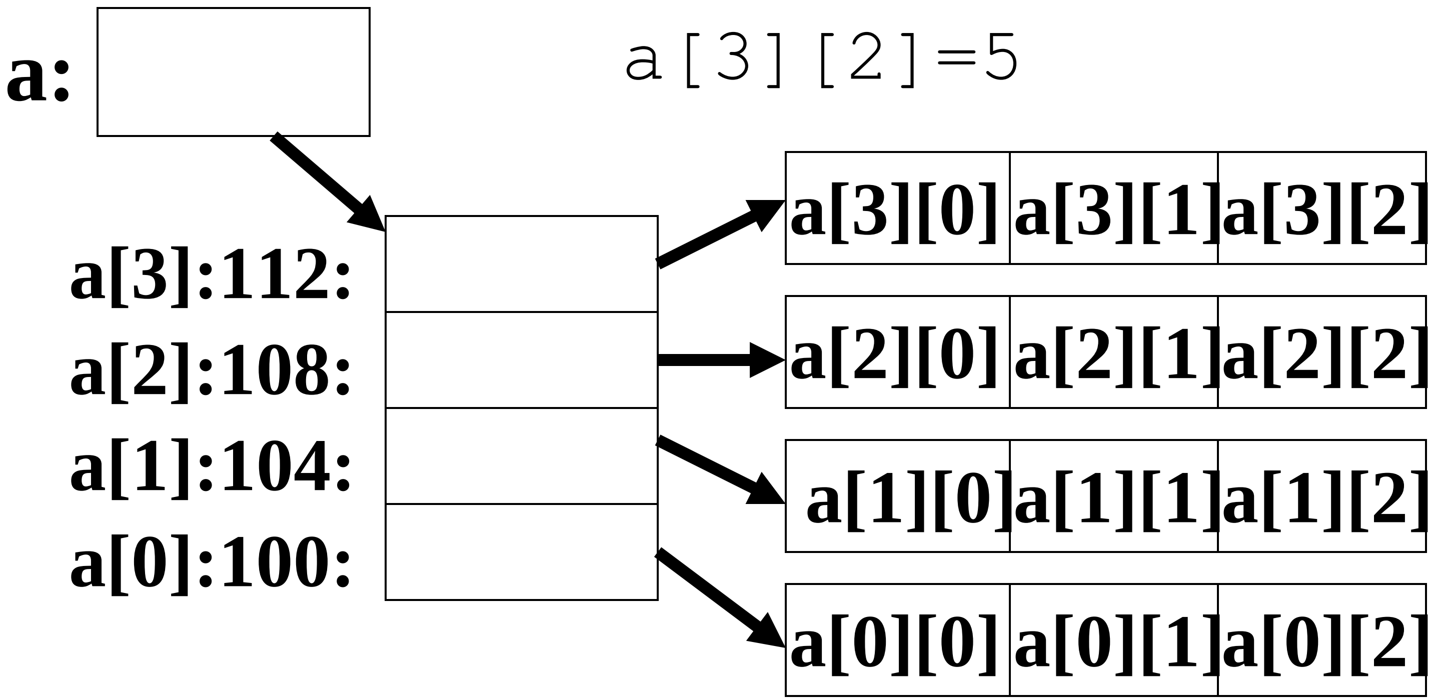
- 3rd approach. a is a pointer to an array of pointers to rows.

```
int **a;
a=(int**)malloc(4*sizeof(int*));
assert( a!= NULL)
for(int i=0; i<4; i++)
{
  a[i]=(int*)malloc(3*sizeof(int));
  assert(a[i] != NULL)
}
```

2D Array 3rd Implementation

- a is a pointer to an array of pointers to rows. (cont.)

```
int **a;
a[3][2]=5
```

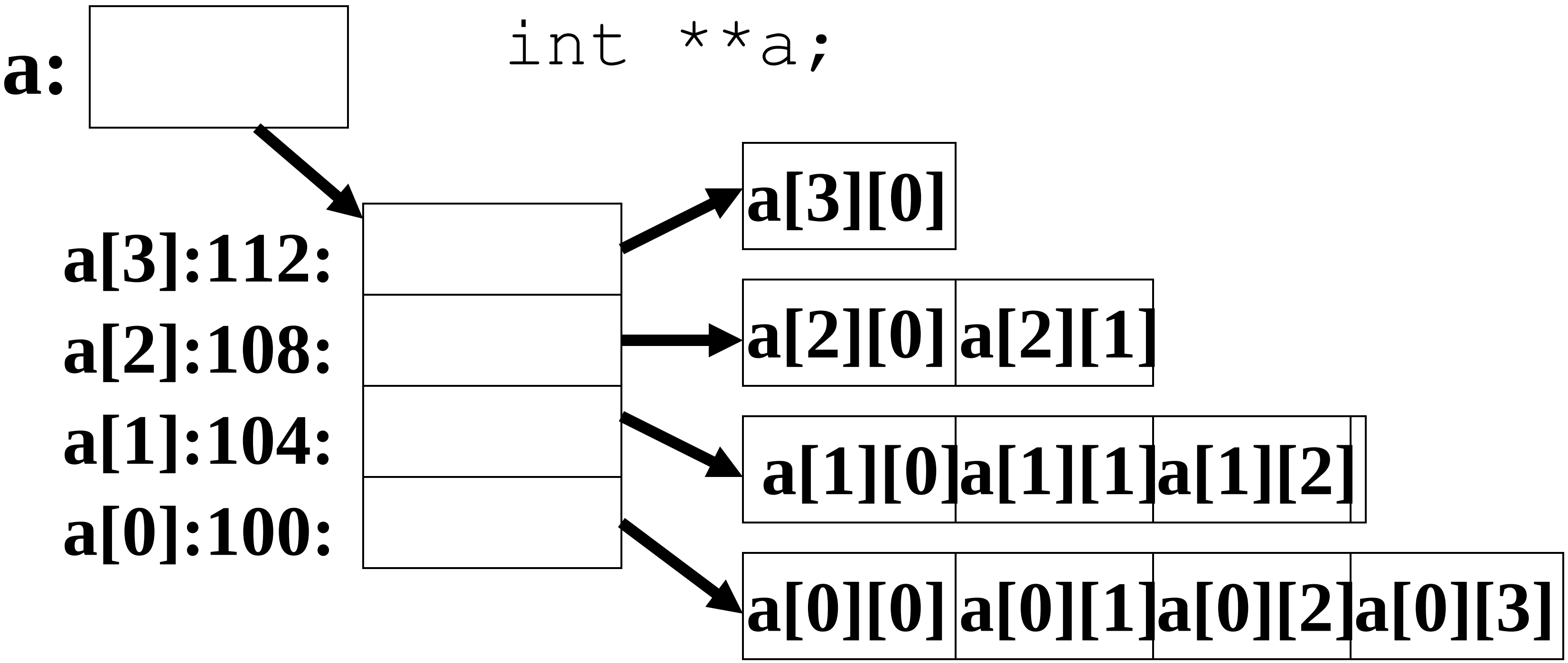


Advantages of Pointer Based Arrays

- You don’t need to know in advance the size of the array (dynamic memory allocation)
- You can define an array with different row sizes
- A jagged array uses multiple small blocks instead of one large block of memory.

Advantages of Pointer Based Arrays

- Example: Triangular matrix



Allocating One block 2d arrays in the heap

- You can also allocate a single block 2D array in the heap.
- You will need helper functions to access it.
- Example for an array 4 x 3

```
int *a = (int *)malloc( 4 * 3 * sizeof(int));

int getVal(int *a, int i, int j) {
      return  a[i * 3 + j];
      // or  return *(int*)((char*)a+i*3*sizeof(int)+j*sizeof(int));
}

void  setVal(int *a, int i, int j, int val) {
       a[i * 3 + j] = val;
      // or *(int*)((char*)a+i*3*sizeof(int)+j*sizeof(int))=val;

}
```

Pointers to Functions

- Pointers to functions are often used to implement Polymorphism in "C".
- ***Polymorphism***: Being able to use the same function with arguments of different types.
- Example of function pointer:

```
typedef void (*FuncPtr)(int a);
```

- `FuncPtr` is a type of a pointer to a function that takes an "`int`" as an argument and returns "`void`".

An Array Mapper

```
typedef void (*FuncPtr)(int a);

void intArrayMapper( int *array, int n, FuncPtr func ) {
 for( int = 0; i < n; i++ ) {
    (*func)( array[ i ] );
    }
}
int s = 0;
void sumInt( int val ){
    s += val;
}
void printInt( int val ) {
    printf("val = %d \n", val);
}
```

Using the Array Mapper

```
int a[ ] = {3,4,7,8};
main( ){
  // Print the values in the array
  int n = sizeof(a)/sizeof(int);
   intArrayMapper(a,n, printInt);

  // Print the sum of the elements in the array
  s = 0;
  intArrayMapper(a, n, sumInt);
  printf("total=%d\", s);
}
```

A More Generic Mapper

```
typedef void (*GenFuncPtr)(void * a);
void genericArrayMapper( void *array,
      int n, int entrySize, GenFuncPtr fun )
{
  for( int i = 0; i < n; i++; ){
     void *entry = (void*)(
           (char*)array + i*entrySize );
     (*fun)(entry);
  }
}
```

Using the Generic Mapper

```
void sumIntGen( void *pVal ){
   //pVal is pointing to an int
  //Get the int val
  int *pInt = (int*)pVal;
  s += *pInt;
}


void printIntGen( void *pVal ){
   int *pInt = (int*)pVal;
   printf("Val = %d \n", *pInt);
}
```

Using the Generic Mapper

```
int a[ ] = {3,4,7,8};
main( ) {
// Print integer values
   s = 0;
  genericArrayMapper( a, sizeof(a)/sizeof(int),
              sizeof(int), printIntGen);

   // Compute sum the integer values
   genericArrayMapper( a, sizeof(a)/sizeof(int),
             sizeof(int), sumIntGen);
   printf("s=%d\n", s);

}
```

Swapping two Memory Ranges

- In the lab1 you will implement a sort function that will sort any kind of array.
- Use the array mapper as model.
- When swapping two entries of the array, you will have pointers to the elements (`void *a, *b`) and the size of the entry `entrySize`.

```
void * tmp = (void *) malloc(entrySize);
assert(tmp != NULL);
memcpy(tmp, a, entrySize);
memcpy(a,b , entrySize);
memcpy(b,tmp , entrySize);
```

- Note: You may allocate memory only once for tmp in the sort method and use it for all the sorting to save muliple calls to malloc. Free tmp at the end.

Generic Sorting Pseudocode

```
mysort(void * array, int ascending, int n,
                   int elemSize, ComFunc comp)
//Assume bubblesort
for(int j=0; ….) {
  for( int i=0; ….) {
     // Get address of item i
     char * p = ((char*) array+i*elemSize);
     //Get address of item i+1
    char *q = ((char*) array+(i+1)*elemSize);
   int result= comp((void*)p,(void*)q;
  //Swap if necessary
}
```

String Comparison in Sort Function

- In lab1, in your sort function, when sorting strings, you will be sorting an array of pointers, that is, of "char* entries.
- The comparison function will be receiving a “pointer to char*” or a” char**” as argument.

```
int StrComFun( void *pa, void *pb) {
   char** stra  =  (char**)pa;
   char ** strb  = (char**)pb;
   return strcmp( *stra, *strb);
}
```

Bits and Bytes

- Bit
  - It stores 1 or 0
- Byte
  - It is a group of 8 bits that can be individually addressable.
- Word
  - It is a group of 4 bytes (32 bit architecture) or
  - It is a group of 8 bytes (64 bit architectures)
  - The address of a word is aligned to either 4 or 8 bytes respectively (multiple of 4 or 8 bytes).

Interpretation of bits

- Sometimes device registers are mapped to memory. This is called Memory Mapped I/O.
- In this case, a bit can represent some value or state of the device:
  - Bit 0 – Printer is on-line/off-line
  - Bit 1 – Landscape/Letter mode
  - Bit 2 – Printer need attention

Interpretation of bits

- Combination of bits are used as integers

0	1	0	1	1	0	0	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

$$2^6 + 2^4 + 2^3 + 2^0 =$$

$$64 + 16 + 8 + 1 = 89$$

Hexadecimal Notation

- Compact form to represent binary numbers
- It uses base 16.
- 4 bits represent an hexadecimal digit

Hex	Binary			
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1

Hex	Binary			
8	1	0	0	0
9	1	0	0	1
A	1	0	1	0
B	1	0	1	1
C	1	1	0	0
D	1	1	0	1
E	1	1	1	0
F	1	1	1	1

Hexadecimal Notation

- Example:
  - Hexadecimal: 0xF4534004
  - Binary:

    1111 0100 0101 0011 0100 0000 0000 0100
  - Hexadecimal

    F 4 5 3 4 0 0 4

Decimal:

$15*16^7 + 4*16^6 + 5*16^5 + 3*16^4 + 4*16^3 + 4*16^0$

Example of Character Encodings

- EBCDIC
- ASCII
- Unicode

EBCDIC

- Extended Binary Coded Decimal Interchange Format
- It was created by IBM in the 1960s
- No longer in use except in some IBM mainframes

ASCII

- American Standard Code for Information Exchange
- Used widely in UNIX and PCs
- It uses 7 bits or 128 values
- It only encodes the English Alphabet

ASCII Table

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

http://www.ascii.ws/ascii-chart.html

UNICODE

- Each character is 16 bits long (2 bytes)
- It is used to represent characters from most languages in the world.
- It is used for internationalization of programs.
- Java and C# use UNICODE to represent strings internally.

Representation of Strings

- In a “C” program a string is a sequence of characters delimited by a null character.

0x48	0x65	0x6c	0x6c	0x6f	0x00				

H	e	l	l	o	\0				

- In PASCAL the first byte represents the length of the string.

0x5	0x48	0x65	0x6c	0x6c	0x6f				

  - Standard strings were limited to a length of 255

Integer Representation in Binary

- Each binary integer is represented in k bits where k is 8, 16, 32, or 64 depending on the type and architecture.

Integer Representation

- Example

10010101 = 1*2^7 + 1*2^4+1*2^2+1*2^0 =

= 128 + 16 + 4 + 1

= 149

Binary Integer Addition

- Same as decimal addition:
- Use S1, S2 and Carry (C) to compute R and next Carry (C+)

```
   00 C   (Carry)
 1011 S1  (11)
+0110 S2  (06)
-----------------
    1 R
```

Truth Table

C	S1	S2	C+	R
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

Binary Integer Addition

```
  100 C   (Carry)
 1011 S1  (11)
+0110 S2  (06)
   01 R
```

Truth Table

C	S1	S2	C+	R
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

Binary Integer Addition

```
 1100  C   (Carry)
 1011  S1  (11)
+0110  S2  (06)
  001  R
```

Truth Table

C	S1	S2	C+	R
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

Binary Integer Addition

```
11100 C   (Carry)
 1011 S1  (11)
+0110 S2  (06)
 0001 R
```

Truth Table

C	S1	S2	C+	R
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

Binary Integer Addition

```
11100 C   (Carry)
 1011 S1  (11)
+0110 S2  (06)
10001 R   (17)
```

Truth Table

C	S1	S2	C+	R
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

Binary Integer Subtraction

- Same as decimal subtraction:
- Use S1, S2 and Carry (C) to compute R and next Carry (C+).

```
    00 C   (Carry)
  1011 S1 (11)
 -0110 S2 (06)
     1 R
```

Binary Integer Subtraction

```
  000 C   (Carry)
 1011 S1  (11)
-0110 S2  (06)
   01 R
```

Truth Table

C	S1	S2	C+	R
0	0	0	0	0
0	0	1	1	1
0	1	0	0	1
0	1	1	0	0
1	0	0	1	1
1	0	1	1	0
1	1	0	0	0
1	1	1	1	1

Binary Integer Subtraction

```
 1000 C   (Carry)
 1011 S1  (11)
-0110 S2  (06)
  101 R
```

Truth Table

C	S1	S2	C+	R
0	0	0	0	0
0	0	1	1	1
0	1	0	0	1
0	1	1	0	0
1	0	0	1	1
1	0	1	1	0
1	1	0	0	0
1	1	1	1	1

Binary Integer Subtraction

```
01000 C   (Carry)
 1011 S1  (11)
-0110 S2  (06)
 ----  --  ----
 0101 R   (05)
```

Truth Table

C	S1	S2	C+	R
0	0	0	0	0
0	0	1	1	1
0	1	0	0	1
0	1	1	0	0
1	0	0	1	1
1	0	1	1	0
1	1	0	0	0
1	1	1	1	1

Binary Multiplication

- Same as decimal multiplication
- Just need to memorize multiplication table for 0 and 1
- Perform sums and shifts iteratively based on the 0/1 of the multiplicator

Binary Multiplication

```
  1011
x  110
------
  0000
```

Binary Multiplication

```
   1011
 x  110
   ----
   0000
 +1011
  -----
  10110
```

Binary Multiplication

```
      1011    (11)
    x  110    ( 6)
    ------    ----
      0000
    +1011
    -----
     10110
   +1011
   -----
   1000010    (64+2=66)
```

Binary Multiplication.

Another example

```
      1001    (9)
    x  101    (5)
      ----    ---
      1001
```

Binary Multiplication.

Another example

```
      1001   (9)
    x  101   (5)
      ----   ---
      1001
    +0000
     -----
     01001
```

Binary Multiplication.

Another example

```
      1001   (9)
    x  101   (5)
      ____   ___
      1001
    +0000
    _____
     01001
  +1001
  _____
    101101   (32+8+4+1=45)
```

Binary Division

- Same as decimal division
- Just need to memorize multiplication table for 0 and 1
- Perform subtractions and shifts iteratively

Binary Division

```
          1
      ________
100 |  10110
      -100
       001
```

Binary Division

```
          10
      ___________
100 | 10110
     -100
      0011
```

Binary Division

```
            101     (5)
       ____________
(4) 100 | 10110 (16+4+2=22)
         -100
          00110
         -  100
          010 (2)
```

Binary Representation of Negative Integer Numbers

- Three representations
  - Sign and Magnitude
  - 1-complement
  - 2-complement

Sign and Magnitude Representation

- 1 bit for sign
- Other bits for the absolute value
- Example:

```
+5 =  0 0000101
-5 =  1 0000101
   sign magnitude
```

1-Complement

- Negative numbers are obtained by inverting all bits.
- Example:

```
+5 =  00000101
-5 =  11111010
```

2-Complement

- Negative numbers are obtained by subtracting 1 from the positive number and inverting the result.
- Example:

```
+5 =  00000101
-5 =  00000101
   = -00000001
   =  00000100
-5 =  11111011
```

```
+5 +(-5):
  00000101
 +11111011
  00000000
(ignoring overflow)
```

2-Complement

- 2-complement representation is widely used because the same piece of hardware used for positive numbers can be used for negative numbers:
- Example:

```
+5 = 00000101
-3  = 00000011
    =-00000001
    = 00000010
-3  = 11111101
```

```
+5 +(-3):
  00000101
 +11111101
  00000010 (2)
(ignoring overflow)
```

Right Shift >> with sign extension.

- When using i >> n (shift right) the behavior will change if i is a signed int or i is unsigned int.
- If i is signed int and i is negative then i >> n will insert *n* 1s in the left side.
- This is called “signed extension”.

Example:

```
int i = 0xFFFFFFFB; // i=11111111 11111111 11111111 11111011;
int j2 = (i >> 2);  // j=11111111 11111111 11111111 11111110 =0xFFFFFFFFE
int k =  0x00000005; // k=00000000 00000000 00000000 00000101
int k2 = (k>>2); // k=00000000 00000000 00000000 00000001
```

- If i is unsigned then i >> n will insert *n* 0s in the left side.

```
unsigned i = 0xFFFFFFFB; // i=11111111 11111111 11111111 11111011
unsigned j = (i >> 2); // j=00111111 11111111 11111111 11111110=3FFFFFFE
```

Bit Operations: Left and Right Shift << >>

- x >> i
  - Shifts bits of a number x to the right i positions
- x << i
  - Shifts bits of a number x to the left i positions
- Example:

```
int i, j;
i = 5; // In binary i is 00000101
j= (5 << 3); // In binary j is 00101000
printf("i=%d j=%d\n"); // Output: i=5 j=40
```

- **`x<<i is equivalent to x*2^i`**
- **`x>>i is equivalent to x/2^i`**

Bitwise Operations: OR |

- The “|” operator executes “OR” bit operation.

```
unsigned x = 0x05; // 00000101
unsigned y = (x | 0x2);
          // 00000101 | 00000010 = 00000111
printf(“x=0x%x 0x%x\n”, x,y); // x=0x5 y=0x7

Example:
/usr/include/unistd.h:
#define O_READ 0x01
#define O_WRITE 0x02
In your code
d = open("/path/to/dir", O_READ| O_WRITE);
// 0x01 | 0x02 == 0x03
```

Bitwise Operations: AND &

- The “&” operator executes “AND” bit operation.

```
unsigned x = 0x05; // 00000101
unsigned y =(x & 0x3);// 00000101 & 00000011 =00000001
printf(“x=0x%x y=0x%x\n”, x,y); // x=0x5 y=0x1
```

Bitwise Operations: XOR ^

- The “^” operator executes “XOR” bit operation.
- XOR : 0^0==0, 0 ^1 == 1, 1^0==1, 1^1==0

```
unsigned x = 0x05; // 00000101
unsigned y =(x ^ 0x3);// 00000101 ^ 00000011 =00000110
printf(“x=0x%x 0x%x\n”, x,y); // x=0x5 y=0x6

Example:
// Inverting the bits of a number
unsigned x = 0x05; // 00000000 00000000 00000000 00000101
unsigned y = x ^ 0xFFFFFFFF;
//0xFFFFFFFF=11111111 11111111 1111111 11111111
        // y = 11111111 1111111 1111111 11111010
```

Bitwise Operations: NOT ~

- The “~” negates bits.

```
unsigned x = 0x05; // 0000000 0000000 0000000 00000101
unsigned y = ~x;
    // ~00000101 = 11111111 11111111 11111111 11111010
printf(“x=0x%x 0x%x\n”, x,y); // x=0x5 y=0xFFFFFFFA
```

Using Bitwise Operations: Test if bit i is set:

```
int i = 4;
unsigned x = 23; // x = 00010111

// Test if bit i is set in x
// Create mask with bit i set.
unsigned mask = (1 << i); // mask == 00010000

// Test if bit i is set
unsigned y = (x & mask); // y = 00010111 & 00010000 =00010000
int bit = (y >> i);       // bit = 00000001
                          // bit i in x is set.
```

Using Bitwise Operations: Set bit i :

```
int i = 3;
unsigned x = 23; // x = 00010111

// Set bit i in x
// Create mask with bit i set.
unsigned mask = (1 << i); // mask == 0001000

// Set bit i
unsigned y = (x | mask); // y = 00010111 | 0001000 = 00011111
```

Using Bitwise Operations: Clear bit i :

```
int i = 2;
unsigned x = 23; // x = 00010111

// Set bit i in x
// Create mask with bit i set.
unsigned mask = (1 << i); // mask == 00000100
unsigned mask0 = ~mask;  // mask0 == 11111011

// Clear bit i
unsigned y = (x & mask0); // y = 00010111 & 11111011 = 00010011
```

Converting Real Numbers to Binary

- 67.24 to binary
- Integer part: 67
  - 67/2=33R1->1, 33/2=16R1 ->1
  - 16/2=8R0->0, 8/2=4R0->0
  - 4/2=2R0->0, 2/2=1R0->0, 1/2=0R1 Stop
  - → 67= 1000011b

Decimal part: 0.24

  - .24*2=0.48->0, 0.48*2=0.96->0,
  - 0.96*2=1.92->1, 0.92*2=1.84->1,
  - 0.84*2=1.68->1, 0.68*2=1.36->1,
  - 0.36*2->0.72, 0.72*2=1.44
  - -> 0.24 = .00111101
  - 67.24= 1000011.00111101b
  - 1000011.0011110x2^0=1.0000110011110x2^6

Floating Point Representation

- Store both the exponent and mantissa
- Example:
  - $3.5\text{x}10^{-16}$
- In binary the representation uses base 2 instead of base 10
- Example:
  - $\mathbf{1.101x2^{-010}}$

Representing Real Numbers

- Two representations:
  - Fixed point numbers
    - Fixed number of bits for both integer and mantissa.

      N.M – N bits for integer part and M bits for mantissa.

      Example: 16 bit real number N =8 bits, M = 8 bits

      Problem: Range of numbers is small. N=8 bits can represent therange -128 to +127

      +127=01111111=+127

      -128=~(10000000-1)=-~(01111111)=10000000=-128
    - Fixed point arithmetic is easier to implement using integer arithmetic
- Floating Point Numbers:
  - The bits allocated for integer part and mantissa is variable. Example= N.M where N+M =16bits
  - Decimal point moves. Improved range but complex

Floating Point Representation

- The most common is the IEEE-754 standard

Float:

bias = 127

Double:

bias = 1023

$$Val = (-1)^{s} \times (1.m) \times 2^{(e-bias)}$$

Notice that the 1 in 1.m is always assumed. The only exception of all the numbers is 0, that is represented with an exponent of 0.

Floating Point Representation Example

- Double value in memory (in hex):

  0x 4025 0000 0000 0000

  Binary:

  0100 0000 0010 0101 0000 0000 0000 0000

  0000 0000 0000 0000 0000 0000 0000 0000

  Decimal?

  s (bit 63) = 0 = positive number

  e (bits 52 to 62) = 100 0000 0010 = 1024 + 2 = 1026

  m (bits 0 to 51) = .0101 0000 0000 0000 0000

  Val = $(-1)^0 \times (1.0101)_2 \times 2^{(1026-1023)}$

  $= 1\text{x} (2^0+2^{-2}+2^{-4})\text{x}2^3=(1+1/4+1/16)\text{x}8=8+2+.5=10.5$

Byte Order

- There are two byte orders:
  - Little Endian – Least significant byte of the integer is in the lowest memory location.
  - Big Endian – Most significant byte of the integer is in the lowest memory location

Representation of 0x00000005

- Little Endian

0	1	2	3
0x05	0x00	0x00	0x00

- Big Endian

0	1	2	3
0x00	0x00	0x00	0x05

Representation of 0x00002167

- Little Endian

0	1	2	3
0x67	0x21	0x00	0x00

- Big Endian

0	1	2	3
0x00	0x00	0x21	0x67

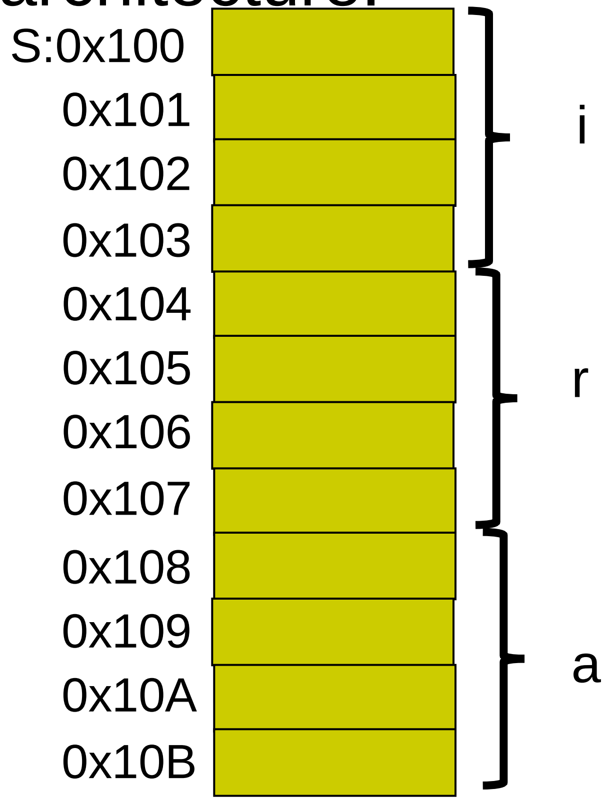
How to know if it is Little or Big Endian

```
int isLittleEndian()
{
    int i = 5;
    char *  p = (char *) &i;
     if (*p==5) {
       return 1;
    }
    return 0;
}
```

Structures

- Structures are a combination in memory of primitive types. Assuming a 32 bit architecture.
- Example:

```
struct {
  int i;
  float r;
  char * a;
} s;
```



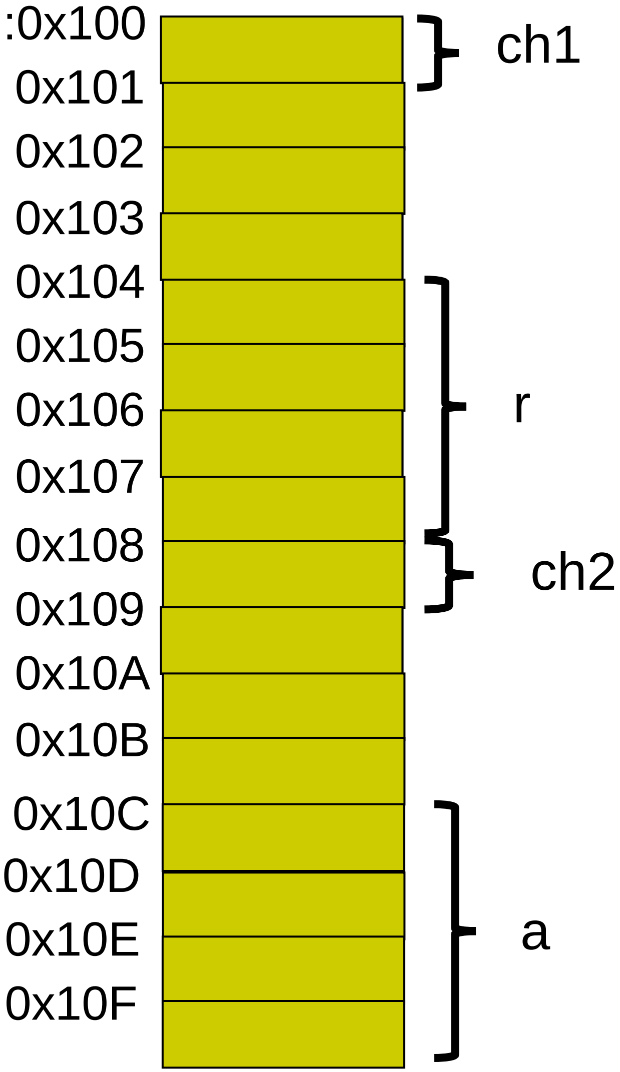
Structures and Alignment

- Integers, floats, and pointers have to be aligned to 4 bytes (in a 32 bit architecture).
  - This means that the memory address have to be a multiple of 4, that is, the last hex digit of the address has to be 0, 4, 8, or C.
- Doubles have to be aligned to 8 bytes.
  - This means that the memory address have to be a multiple of 8, that is, the last hex digit of the address has to be 0, or 8.
- If they are not aligned, the CPU will either get an “bus error” (ARM) or slow down the execution when trying to access this data (Intel).

Example of Alignment in Structures (32 bit arch)

- Example:

```
struct {
   char ch1;
   int r;
   char ch2;
   char * a;
} x;
```

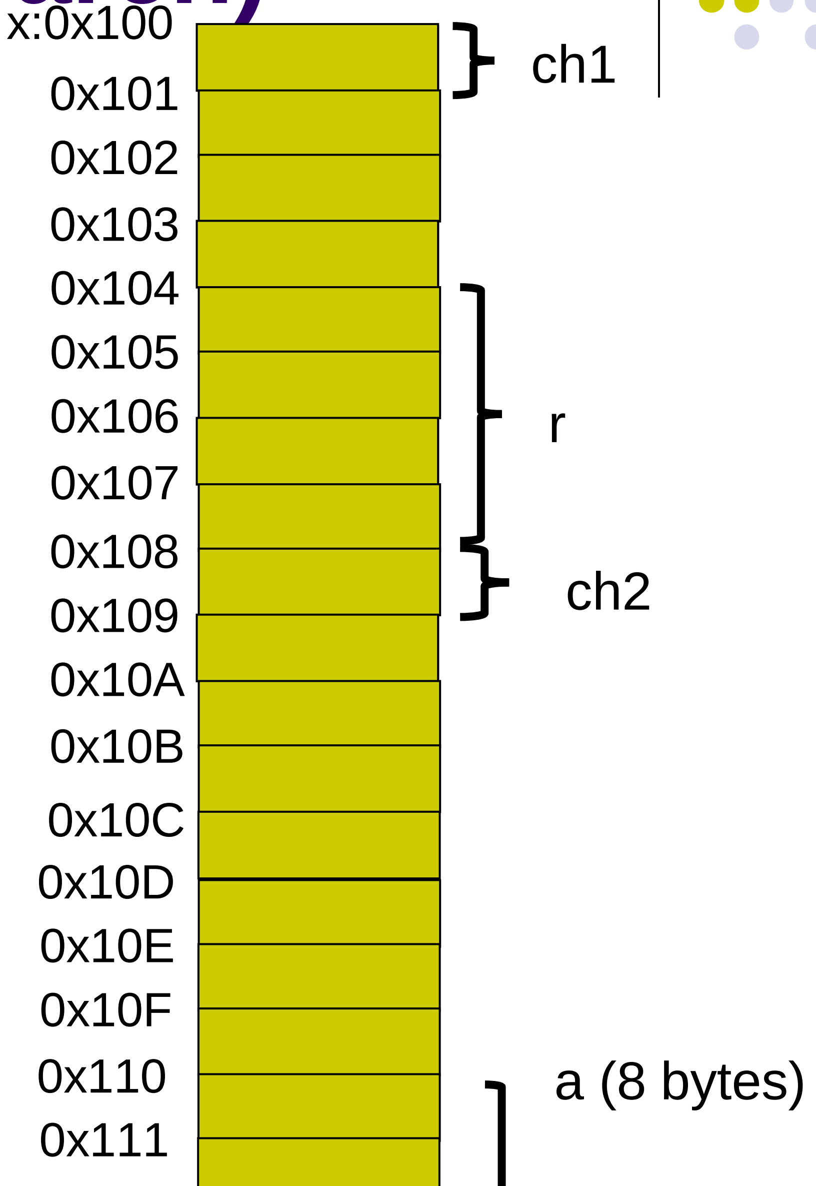


Example of Alignment in Structures (64 bit arch)

- Example:

```
struct {
   char ch1;
   int r;
   char ch2;
   char * a;
} x;
```

sizeof(x)==24



Project 2- Memory Dump

```
grr@data:~/cs250/Fall2023/project2-src$ ./mem
&head=0x7fff7d2861b8
head=0x560f7bb816b0
-----
0x0000560F7BB816B0: D0 16 B8 7B 0F 56 00 00 F0 16 B8 7B 0F 56 00 00 ...{.V.....{.V..
0x0000560F7BB816C0: 00 00 00 00 00 00 00 00 21 00 00 00 00 00 00 00 ........!.......
0x0000560F7BB816D0: 57 65 6C 63 6F 6D 65 20 00 00 00 00 00 00 00 00 Welcome ........
0x0000560F7BB816E0: 00 00 00 00 00 00 00 00 21 00 00 00 00 00 00 00 ........!.......
0x0000560F7BB816F0: 10 17 B8 7B 0F 56 00 00 30 17 B8 7B 0F 56 00 00 ...{.V..0..{.V..
0x0000560F7BB81700: 00 00 00 00 00 00 00 00 21 00 00 00 00 00 00 00 ........!.......
0x0000560F7BB81710: 74 6F 20 00 00 00 00 00 00 00 00 00 00 00 00 00 to .............
0x0000560F7BB81720: 00 00 00 00 00 00 00 00 21 00 00 00 00 00 00 00 ........!.......
0x0000560F7BB81730: 50 17 B8 7B 0F 56 00 00 00 00 00 00 00 00 00 00 P..{.V..........
0x0000560F7BB81740: 00 00 00 00 00 00 00 00 21 00 00 00 00 00 00 00 ........!.......
0x0000560F7BB81750: 63 73 32 35 30 00 00 00 00 00 00 00 00 00 00 00 cs250...........
0x0000560F7BB81760: 00 00 00 00 00 00 00 00 A1 08 02 00 00 00 00 00 ................
0x0000560F7BB81770: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x0000560F7BB81780: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x0000560F7BB81790: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

```
head = (struct List *) malloc(sizeof(struct List));
head->str=strdup("Welcome ");
head->next = (struct List *) malloc(sizeof(struct List));
head->next->str = strdup("to ");
head->next->next = (struct List *) malloc(sizeof(struct List));
head->next->next->str = strdup("cs250");
head->next->next->next = NULL;
```

IV. Variety of Processors

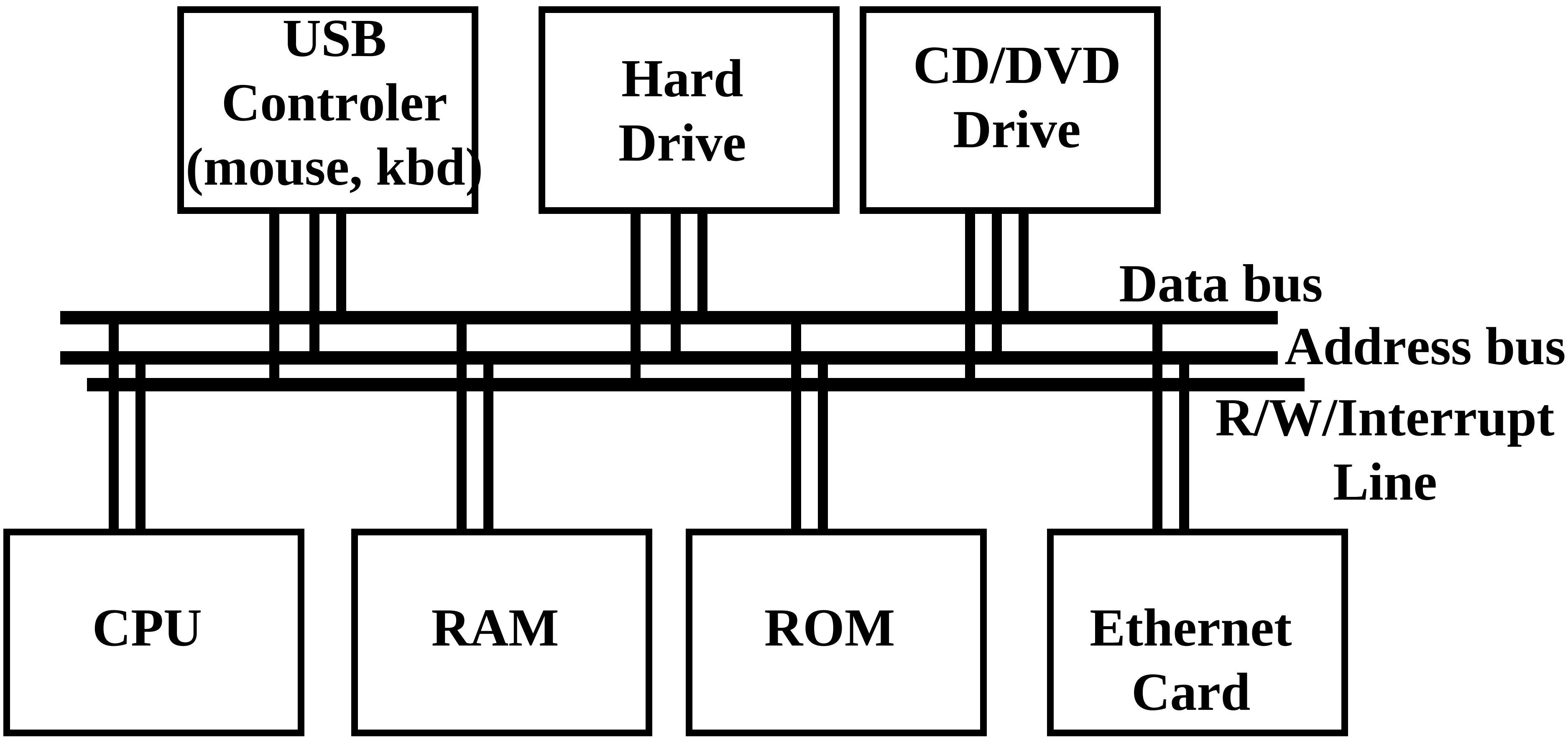
Von Neumann Architecture

- Modern processors follow this design
- Programs are stored in memory, in the same way data is stored in memory.
- In the early days, before the “Stored Program” concept, computers had to be “rewired” in order to run a different program.
- In those old days, often took weeks to load a different program.

Von Neumann Architecture

- A computer has an ***address bus*** and a ***data bus*** that are used to transfer data from/to the CPU, RAM, ROM, and the devices.
- The CPU, RAM, ROM, and all devices are attached to this bus.

Von Newman Architecture



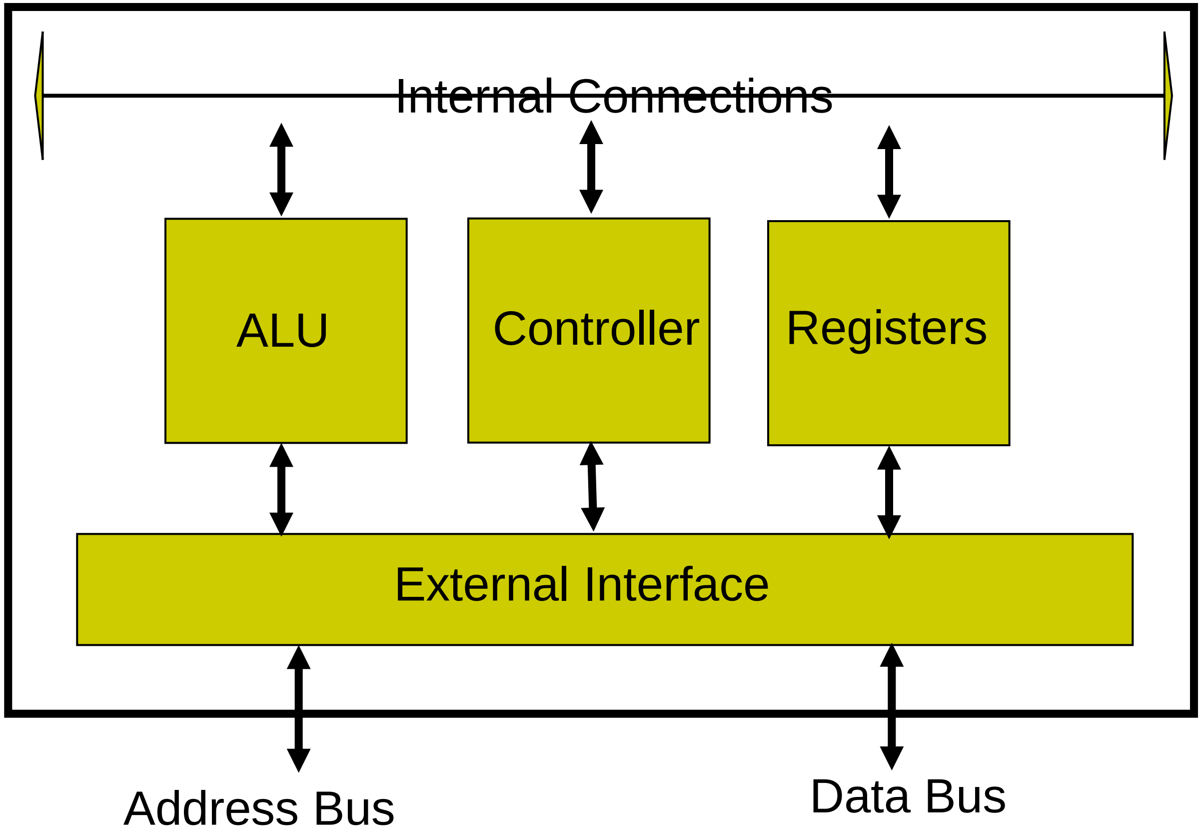
Processors

- Digital device that performs computation using multiple steps.
- Types of Processors:
  - Fixed Logic – Least powerful. Single Operation.
  - Selectable Logic – Performs more than one operation.
  - Parameterized Logic Processor – Accepts a set of parameters in the computation.
  - Programmable Logic Processor – Greatest Flexibility. Function to compute can be changed. CPU's belong to this type of processors.
- CPU – Central Processing Unit

Components of a CPU

- Controller
- ALU – Arithmetic and Logical Unit
- Registers - Local Data Storage
- Internal Interconnections
- External Interface

Components of the CPU



Components of the CPU

- Controller
  - Controls the execution
  - Initiates the sequence of steps
  - Coordinates other components
- ALU – Arithmetic and Logical Unit
  - It provides the Arithmetic and Boolean Operations.
  - It performs one operation at a time.

Components of the CPU

- Registers
  - Holds arguments and results of the operations
- Internal Connections
  - Transfers values across the components in the CPU.
- External Interface
  - Provides connections to external memory as well as I/O devices

ALU – Arithmetic Logic Unit

- It is the part of the CPU that performs the Arithmetic and Boolean operations
  - Integer Arithmetic - add, subtract, multiply, divide
  - Shift - left, right, circular
  - Boolean - and, or, not, exclusive or

Processor Categories

- Coprocessors
  - Operates in conjunction with other processor. Example: Floating Point Accelerator.
- Microcontroller
  - Small programmable device. Dedicated to control a physical system. Example: Electronic Toys.
- Microsequencer
  - Use to control coprocessors, memory and other components inside a larger processor board.

Processor Categories

- Embedded System Processor
  - It is able to run sophisticated tasks
  - More powerful than a microcontroller
  - Example: The controller in a an MP3 player that includes User Interface and MP3 decoding.
- General Purpose Processor
  - Most powerful type of processor
  - Completely Programmable
  - Example: Pentium processor

Evolution of Processor Technologies

- Discrete Logic
  - Use TTL Gates etc used to implement processor.
  - It could use multiple boxes and circuit boards.
- Single circuit board
  - Multiple chips/controllers in a single board.
- Single chip
  - All the components are in a single chip.

Fetch-Execute Cycle

- This is the basics for programmable processors.
- It allows moving through the program steps a

while (1) {

Fetch from memory the next instruction to execute in the program.

Execute this instruction.

}

Clock Rate and Instruction Rate

- Clock rate
  - It is the rate at which gates and hardware components are clocked to synchronize data transfer.
- Instruction rate
  - It is the time required to execute an instruction.
  - Different instructions may take different times.
  - Example: Multiplication and division will take more clock cycles than addition and subtraction.

Starting a Processor

- When the CPU is powered on or when reset
  - The CPU is initialized
  - The fetch-execute cycle starts.
  - The first instruction to execute will be in a known memory location, E.g. 0x1000
  - This process is called “bootstrap”.

Stopping a Processor

- When the application finishes or it is waiting for an event,
  - The program may enter an infinite loop.
  - In an OS, that infinite loop is often called
    - “Null Process” or
    - “System Idle Process”.

V. Processor Types and Instruction Sets

How to Choose an Instruction Set

- A small set is easy to implement but inconvenient for programmers.
- A large set is convenient for programmers but expensive to implement.
- When designing an instruction set we need to consider
  - Physical size of the Processor
  - How the processor will be used
  - Power consumption

Parts of an Instruction

- Opcode
  - Specifies the instruction to be executed
- Operands
  - Specifies the registers, memory location, or constants used in the instruction
- Result
  - Specifies the registers or memory location where the result of the operation will be placed.

Opcode	Operand1	Operand2	Result

Instruction Length

- Fixed Length
  - Every instruction has the same length
  - Reduces the complexity of the hardware
  - Potentially, the program will run faster.
- Variable Length
  - Some instructions will take more space than others
  - It is appealing to Assembly code programmers (Not a very strong advantage. Most programs are written in a high-level language).
  - More efficient use of memory.
  - Pentium continues using variable length instructions because of *backward-compatibility* issues.

General Purpose Registers

- They are used to store operands and results
- Each register has a small size: 1 byte, 4 bytes, or 8 bytes.
- *Floating Point Registers*
  - *Special registers used to store floating point numbers.*

Example of Using Registers

- Load A from location 0x100 and B from location 0x104. Store A+B in C in location 0x108 (C=A+B);

```
load r1, @0x100
load r2, @0x104
add r1, r2, r3
store r3, @0x108
```

- Register Spilling – Save registers in memory for later use. The number of registers is limited, so very often it is necessary to use memory or the stack to store temporal values.

- Register allocation. Choose what values to keep in the registers instead of memory.

Types of Instruction Sets

- CISC
  - Complex Instruction Set Computer
- RISC
  - Reduced Instruction Set Computer

CISC Instruction Set

- It contains many instructions, often hundreds.
- Some instructions take longer than others to complete
- Variable Length Instructions
- Examples:
  - Move a range of bytes from one place in memory to another
  - Compute the length of a string
- Example: x86

RISC Instruction Set

- It contains few instructions 32 or 64
- Instructions have a fixed length
- Most instructions are executed in one clock cycle.
- Example: Sparc, Alpha, MIPS, ARM

X86-64 Assembly Language

History

- Created by AMD to extend the x86 architecture to use 64 bits
- X86-64 is a superset of x86-32
- It has been adopted by Intel
- It provides an incremental evolution to migrate from x86-32 bits to x86-64 bits.

Register Assignment

Register	Usage	Preserved across function calls
%rax	temporary register; with variable arguments passes information about the number of vector registers used; 1st return register	No
%rbx	callee-saved register	Yes
%rcx	used to pass 4th integer argument to functions	No
%rdx	used to pass 3rd argument to functions; 2nd return register	No
%rsp	stack pointer	Yes
%rbp	callee-saved register; optionally used as frame pointer	Yes
%rsi	used to pass 2nd argument to functions	No
%rdi	used to pass 1st argument to functions	No
%r8	used to pass 5th argument to functions	No
%r9	used to pass 6th argument to functions	No
%r10	temporary register, used for passing a function's static chain pointer	No
%r11	temporary register	No
%r12-r14	callee-saved registers	Yes
%r15	callee-saved register; optionally used as GOT base pointer	Yes
%xmm0–%xmm1	used to pass and return floating point arguments	No
%xmm2–%xmm7	used to pass floating point arguments	No
%xmm8–%xmm15	temporary registers	No
%mmx0–%mmx7	temporary registers	No
%st0,%st1	temporary registers; used to return `long double` arguments	No
%st2–%st7	temporary registers	No
%fs	Reserved for system (as thread specific data register)	No
mxcsr	SSE2 control and status word	partial
x87 SW	x87 status word	No
x87 CW	x87 control word	Yes

Register View

63	31	15	8 7	0	
%rax	%eax	%ax	%ah	%al	Return value
%rbx	%ebx	%ax	%bh	%bl	Callee saved
%rcx	%ecx	%cx	%ch	%cl	4th argument
%rdx	%edx	%dx	%dh	%dl	3rd argument
%rsi	%esi	%si		%sil	2nd argument
%rdi	%edi	%di		%dil	1st argument
%rbp	%ebp	%bp		%bpl	Callee saved
%rsp	%esp	%sp		%spl	Stack pointer
%r8	%r8d	%r8w		%r8b	5th argument
%r9	%r9d	%r9w		%r9b	6th argument
%r10	%r10d	%r10w		%r10b	Reserved
%r11	%r11d	%r11w		%r11b	Used for linking
%r12	%r12d	%r12w		%r12b	Unused for C
%r13	%r13d	%r13w		%r13b	Callee saved
%r14	%r14d	%r14w		%r14b	Callee saved
%r15	%r15d	%r15w		%r15b	Callee saved

Using Argument Registers

- A function can use any of the argument registers also as temporal variables.
- You just have to be aware that after calling a function like printf, the register arguments may be modified.
- If you need these registers after the function returns, save them in the stack before calling a function like printf and restore them after it returns.
- Eg:

```
myfunc:
    …
    pushq %rdi  # Save register because printf may modify it
    pushq %rdi  #Push twice to keep top of stack
                #aligned to 16 bytes
    call printf
    popq %rdi
```

Using Callee-Saved Registers

- If a function uses any of the “callee saved”, registers, it means that their values need to be preserved when the function returns.
- The function has to save them in the stack at the beginning of your function and then restore them before returning.
- Eg:

```
myfunc:
  pushq %rbx # Beginning of myfunc. Save value
  …. Body of myfunc … Use %rbx
  popq %rbx # End of myfunc. Restore value
  ret
```

X86-64 C-Types

C declaration	Intel data type	GAS suffix	x86-64 Size (Bytes)
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double word	`l`	4
`unsigned`	Double word	`l`	4
`long int`	Quad word	`q`	8
`unsigned long`	Quad word	`q`	8
`char *`	Quad word	`q`	8
`float`	Single precision	`s`	4
`double`	Double precision	`d`	8
`long double`	Extended precision	`t`	16

(Bryant/O'Hallaron "x86-Machine Level Programming")

Addressing modes

- Immediate Value

```
movq $0x501208,%rdi  #Put in register %rdi the
                     # constant 0x501208
                     #  %rdi = 0x501208
```

- Direct Register Reference

```
movq %rax,%rdi       #Move the contents of
                     #register %rax to %rdi
                     #  %rdi = %rax
```

- Indirect through a register

```
movq %rsi,(%rdi)#Store the value in %rsi
                #in the address contained in %rdi
                # *(%rdi) = %rsi
```

- Direct Memory Reference

```
movq 0x501208,%rdi #Fetch the contents in memory
                   #at address 0x501208 and store it
                   #in %rdi
                   # %rdi = *(0x501208)
```

Example: Adding two numbers

```
            .text
sum:                                    # long sum(long a, long b) {
            pushq    %rbp               # Save frame pointer
            movq     %rsp, %rbp         #

            movq     %rdi, %rax         #   // a=%rdi b=%rsi ret=%rax
            addq     %rsi, %rax         #   return a + b ;

            leave
            ret                         # }

str1:
            .string   "5+3=%ld\n"

.globl main
main:                                   # main()
                                        #
            pushq    %rbp               # Save frame pointer
            movq      %rsp, %rbp        #

            movq      $3, %rsi          # {
            movq      $5, %rdi          #   // r = %rax
            call      sum               #   long r = sum(5, 3)

            movq      %rax, %rsi        #
            movq      $str1, %rdi       #
            movq      $0, %rax          #   // printf needs 0 in %rax
            call      printf            #   printf("5+3=%ld\n", r);

            leave
            ret                         # }
```

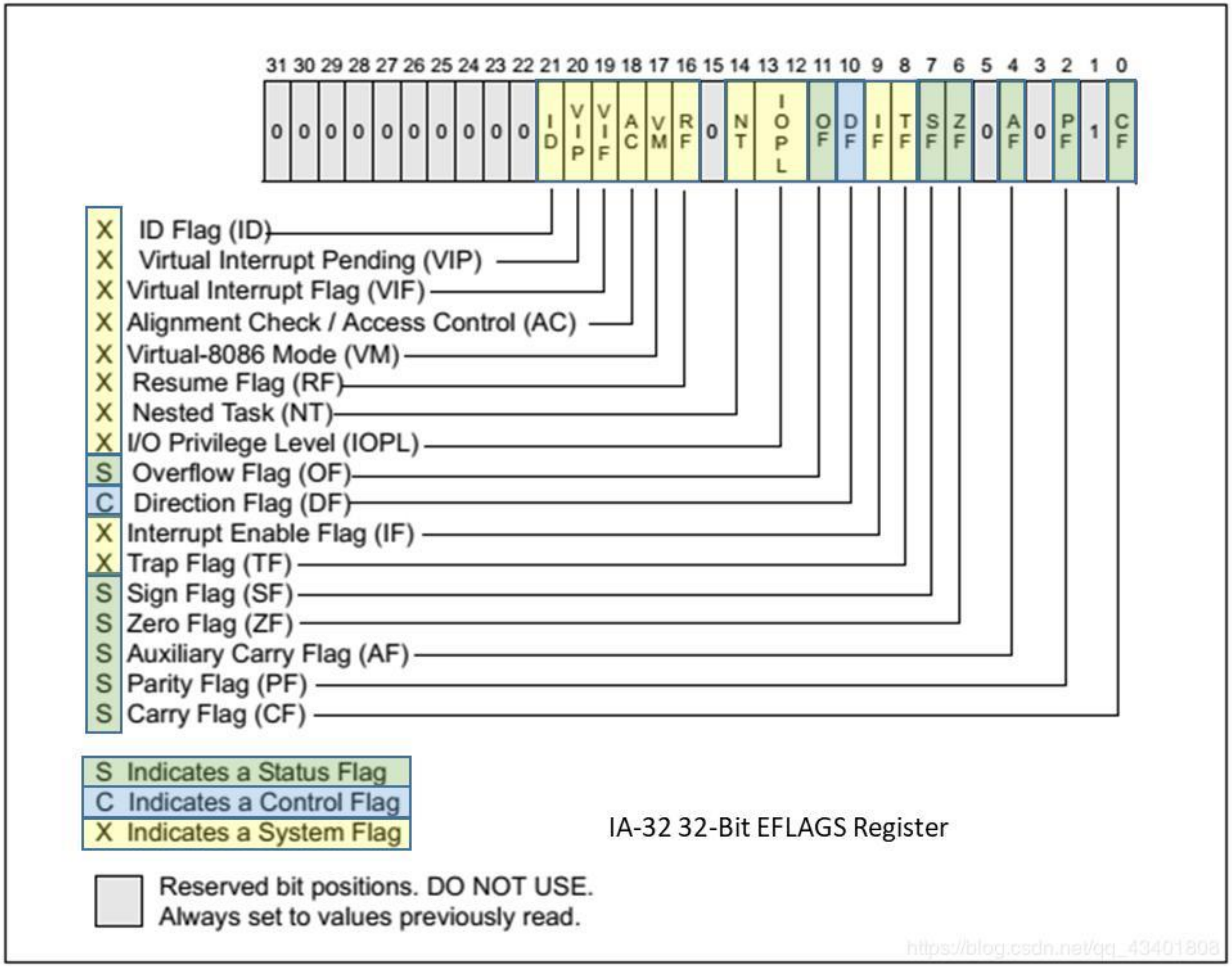
Assembling and running

- To assemble and run program:

```
$data ~/cs250 $ gcc -static -g -o t1 t1.s
$data ~/cs250 $ ./t1
5+3=8
```

- Notice that in the previous example we use quad words during the arithmetic.
- Most of the time there is no penalty for doing that and it makes programs simpler.

X86-64 Flags Register



Using flow control

- To test the difference conditions use:

cmpq S2, S1 # S1 – S2: Compare quad words

or

testq S2, S1 # S1 & S2: Test Quad Word

Conditional Jump Instructions

JE	Jump if equal to zero		ZF = 1
JNE	Jump if not equal to zero		ZF = 0
JGE	Jump if greater or equal		SF = OF
JNGE	Jump if not greater or equal		

Example of if statement: Obtaining maximum of two numbers

```
long max(long a, long b)
{
        long result;
        if (a > b) {
                result = a;
        }
        else {
                result = b;
        }
        return result;
}
```

Example of "if" statement: Obtaining maximum of two numbers

```
        .text
.globl max

max:
                                        #
        pushq    %rbp                   # Save frame pointer
        movq     %rsp, %rbp            #

        cmpq    %rsi, %rdi              # if (a>b)   a:%rdi  b:%rsi (a-b)
        jle     else_branch
        movq    %rdi, %rax              #   result = a
        jmp     end_max

else_branch:                           # else
        movq    %rsi, %rax             #   result = b

end_max:
        leave
        ret                            # return result
```

Example of “while” statement: Obtaining the maximum of an array of numbers.

```
// Finds the max value in an array
long maxarray(long n, long *a) {
        long i=0;
        long max = a[0];
        while (i<n) {
                if (max < *a) {
                        max = *a;
                }
                i++ ;
                a++ ;
        }
        return max;
}
```

Example of "while" statement: Obtaining the maximum of an array of numbers.

```
maxarray.s
          .text
.globl maxarray                          # long maxarray(long n, long *a)
                                         #    // n = %rdi      a = %rsi
maxarray:                                #    // i = %rdx      max = %rax
                                         #
          pushq    %rbp                  # Save frame pointer
          movq     %rsp, %rbp            #
                                         #
          movq     $0,%rdx               #    i=0 ;
          movq     (%rsi),%rax           #    max = a[0];
                                         #
while:    cmpq     %rdx,%rdi             #    while (i<n) { // (n-i>0)
          jle      afterw                #
                                         #
          cmpq     (%rsi),%rax           #      if (max < *a) { // (max-*a<0)
          jge      afterif               #
          movq     (%rsi),%rax           #        max = *a
                                         #      }
afterif:                            #
          addq     $1,%rdx               #      i++ ;
          addq     $8,%rsi               #      a++ ;
          jmp       while                #    }
afterw:   leave
          ret                            # return max; }
```

Example of “while” statement: Obtaining the maximum of an array of numbers using Array Dereferencing

```
// Finds the max value in an array
long maxarray(long n, long *a) {
        long i=0;
        long max = a[0];
        while (i<n) {
                if (max < a[i]) {
                        max = a[i];
                }
                i++ ;
        }
        return max;
}
```

Same program using array dereferencing

```
                        .text
  .globl maxarray                           # // Finds the max value in an array
                                            #
                                            # long maxarray(long n, long *a)
                                            #    // n = %rdi      a = %rsi
  maxarray:                                 #    // i = %rdx      max = %rax
                                            #
          pushq    %rbp                     # Save frame pointer
          movq     %rsp, %rbp               #

          movq    $0,%rdx                   #    i=0 ;
          movq    (%rsi),%rax               #    max = a[0]
                                            #
  while_1:  cmpq     %rdx,%rdi              #    while (i<n) { // (n-i>0)
            jle      afterw                 #
                                            #              //*(long*)((8*i+(char*)a)
            movq     %rdx,%rcx              #      long *tmp = &a[i]; // tmp: %rcx
            imulq    $8,%rcx                #
            addq     %rsi,%rcx              #
                                            #
            cmpq     (%rcx),%rax            #      if (max < *tmp) { // (max-*tmp<0)
            jge      afterif                #
            movq     (%rcx),%rax            #        max = *tmp
                                            #      }
  afterif:  addq     $1,%rdx                #      i++ ;
            jmp      while_1                #
                                            #    }
  afterw:   leave
            ret                             # }
```

Running the program

```
maxarray.c:
#include <stdio.h>

long a[] = {4, 6, 3, 7, 9, 5 };
long maxarray(long n, long *a);

int
main()
{
	printf("maxarray(6,a)=%ld\n", maxarray(6,a));
}

grr@sslab01 ~/cs250 $gcc -g -static -o maxarray maxarray.c
maxarray.s
grr@sslab01 ~/cs250 $ ./maxarray
maxarray(5,a)=9
grr@sslab01 ~/cs250 $
```

Defining Global Variables in Assembly Language

- To create space for a global variable in assembly language use:

```
.data
.comm <var-name>, <data-size>[,<alignment>]
```

where

<var-name> = variable name
<data-size> = Size of variable in bytes
<alignment> = Optional alignment. Address of variable will be a multiple of alignment. Otherwise alignment will be a power of 2 larger to data-size up to 32.

- Example:

```
.data
.comm   a,8               # long a;
.comm   array,40          # long a[5];
.comm   darray, 80,8      # double darray[10];
```

Example Using global vars

```
    # Define global variable a in data section
    .data
    .comm   a,8                    # long a;

    .text
format1:
    .string"a="

format2:
    .string"%ld"

format3:
    .string"a is %ld\n"
.globl main
main:                                               # main()
    pushq    %rbp                                 # Save frame pointer
     movq     %rsp, %rbp                          #

     movq   $format1, %rdi                        #   printf("a=");
    movq    $0, %rax                              #
    call    printf                                #

    movq    $format2, %rdi                        #   scanf("%ld",&a);
    movq    $a, %rsi                              #
    movq    $0, %rax                              #
    call    scanf                                 #

    movq    $format3, %rdi                        #   printf("a=%ld\n",a);
    movq    $a, %rsi                              #
    movq    (%rsi),%rsi                           #
    movq    $0, %rax                              #
    call    printf                                #

    leave
    ret                        # }
```

Notes about using registers

- If you use register arguments to store temporal values in your computations, remember they can be overwritten when you make a function call.

  Example:
  Store something into %rsi, %rdi or any other register argument.
  call printf # This may overwrite %rsi, %rdi or other reg arguments.

- Solutions:
  - 1. Push register arguments before call and pop them after the call. Make sure to push in pairs to keep the stack aligned to 16 bytes.
  - 2. Do not use register arguments. Use callee saved instead. Just push the old value before function starts and pop them when function returns.

```
maxarray:                       #    Beginning of function
          pushq    %rbp          # Save frame pointer
          movq     %rsp, %rbp    #
          pushq    %r13
          pushq    %r14          # (IMPORTANT:16 bytes alignment in stack)
                                 #
           ….. #Use %r12 and %r13. You may call functions and if they
               #modify them, they will restore them before returning.
          popq     %r14
          popq     %r13
          leave
          ret
```

Using Byte registers

- The size of the registers in x86-64 is 8 bytes long.
- However, sometimes we need to do byte manipulations.
- The lowest byte of every register in the x86-64 can be used as a byte register.
- For example, the least significant byte of %r8 is called %r8b
- The least significant byte of %rax is %al
- To use these registers change "q" in the instructions to "b"

  movq %rax, %r8 -> movb %al, %r8b

63	31	15	8 7	0	
%rax	%eax %ax	%ah		%al	Return value
%rbx	%ebx %ax	%bh		%bl	Callee saved
%rcx	%ecx %cx	%ch		%cl	4th argument
%rdx	%edx %dx	%dh		%dl	3rd argument
%rsi	%esi %si			%sil	2nd argument
%rdi	%edi %di			%dil	1st argument
%rbp	%ebp %bp			%bpl	Callee saved
%rsp	%esp %sp			%spl	Stack pointer
%r8	%r8d %r8w			%r8b	5th argument
%r9	%r9d %r9w			%r9b	6th argument
%r10	%r10d %r10w			%r10b	Reserved
%r11	%r11d %r11w			%r11b	Used for linking
%r12	%r12d %r12w			%r12b	Unused for C
%r13	%r13d %r13w			%r13b	Callee saved
%r14	%r14d %r14w			%r14b	Callee saved
%r15	%r15d %r15w			%r15b	Callee saved

Using the stack

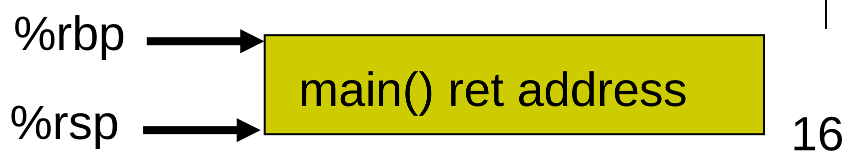
- The stack is used to
  - store the return address
  - store local variables
  - Save registers when running out of them (register spill).
  - pass arguments when they don’t fit in the registers.

Example of Using Stack

```
long sum(long a, long b)
{
        long tmp1 = a;
        long tmp2 = b;
        long result = tmp1 + tmp2;

        return result;
}.

main()
{
        long result = sum(5,3);
        printf("sum(5,3)=%ld\n", result);
}
```

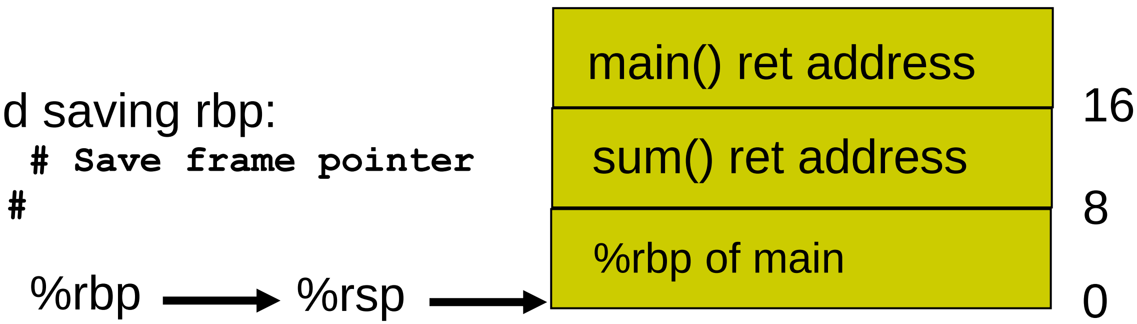
Stack Layout

Before calling sum:
(%rbp== Frame Pointer)

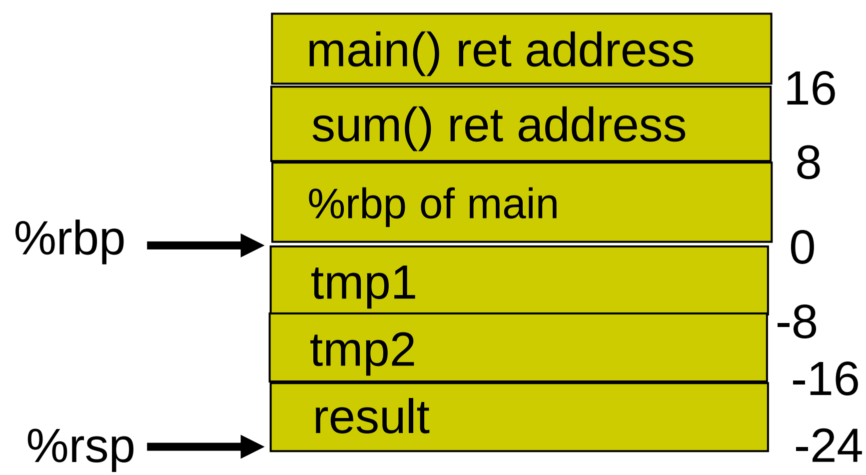


After calling sum and saving rbp:

```
pushq   %rbp        # Save frame pointer
movq    %rsp, %rbp #
```



In sum, after subq $24, %rsp:



Example of Using Stack

```
        .text
.globl sum
        .type   sum, @function
sum:
                                        #
        pushq   %rbp                    # Save frame pointer
        movq    %rsp, %rbp              #

        subq    $24, %rsp               # Create space in stack for
                                        # tmp1, tmp2 and result

        movq    %rdi, -8(%rbp)          # tmp1 = a
        movq    %rsi, -16(%rbp)         # tmp2 = b

        movq    -8(%rbp), %rax
        addq    -16(%rbp), %rax
        movq    %rax, -24(%rbp)         # result = tmp1 + tmp2 ;

        movq    -24(%rbp), %rax         # return result ;

        leave                           # eq. mov %rbp,%rsp
                                        #     pop %rbp

        ret
```

Using gdb with assembly programs

- Use the following instructions to debug assembly programs:
  - stepi – steps in the next instruction. If this is a “call” instruction, it steps in the called function.
  - nexti – Executes next instruciton. It does not enter into a called funciton.
  - disassemble function/label– disassembles the current function or label
  - Break function – Sets a break point in a function
  - Run – run to completion or until a breakpoint
  - disas /r - Disassembles with hexadecimal byte values and assembly codes

Using gdb

```
(gdb) break main
Breakpoint 1 at 0x4004f4
(gdb) run
Starting program: /u/u3/grr/cs250/max
warning: no loadable sections found in added symbol-file system-supplied DSO at 0x7ffff01fe000

Breakpoint 1, 0x00000000004004f4 in main ()
(gdb) stepi
0x00000000004004f9 in main ()
(gdb)
0x00000000004004fe in main ()
(gdb)
0x0000000000400503 in main ()
(gdb)
0x000000000040051c in maxarray ()
(gdb)
0x0000000000400523 in maxarray ()
(gdb) disassemble
Dump of assembler code for function maxarray:
0x000000000040051c <maxarray+0>:        mov    $0x0,%rdx
0x0000000000400523 <maxarray+7>:        mov    (%rsi),%rax
End of assembler dump.
(gdb)
```

Execution Pipeline

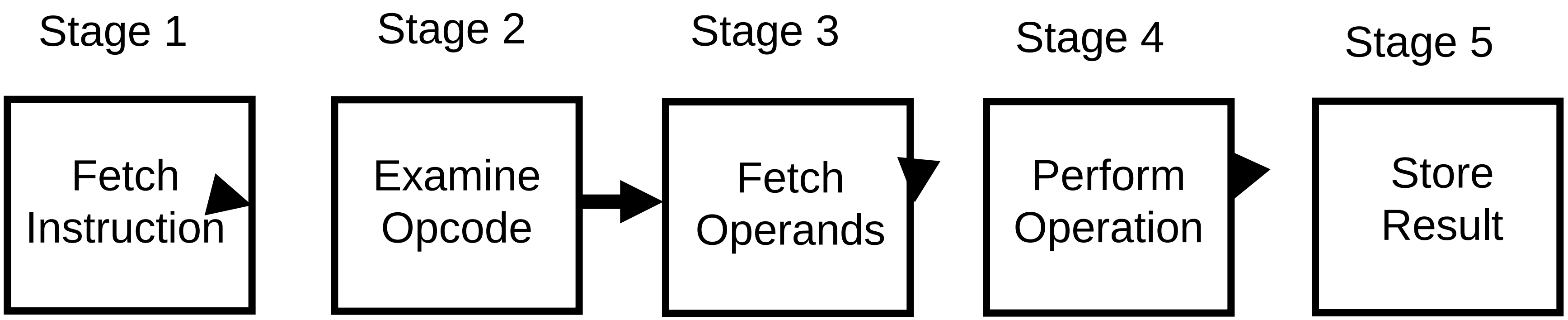
Execution Pipeline

- Hardware optimization technique
- Allows the execution of instructions in parallel.
- Used by RISC architectures

Execution Pipeline

- An instruction is executed by the following steps:
  - Fetch the next instruction
  - Examine the opcode to determine the operands needed.
  - Fetch the operands
  - Perform the specified operation
  - Store the result in the indicated location
- Pipelining executes this steps in parallel for multiple instructions.

Execution Pipeline



Execution Pipeline

- Each stage operate in parallel with a different instruction.
- As a result, an N stage pipeline operates over N instructions simultaneously.
- Each stage takes one clock cycle.
- Each instruction takes one clock cycle once the pipeline is full.

Pipeline Example

Clock	Stage1	Stage2	Stage3	Stage4	Stage5
1	Inst1				
2	Inst2	Inst1			
3	Inst3	Inst2	Inst1		
4	Inst4	Inst3	Inst2	Inst1	
5	Inst5	Inst4	Inst3	Inst2	Inst1
6	Inst6	Inst5	Inst4	Inst3	Inst2
7	Inst7	Inst6	Inst5	Inst4	Inst3
8	Inst8	Inst7	Inst6	Inst5	Inst4
9	Inst9	Inst8	Inst7	Inst6	Inst5

Pipeline Control

- The pipeline is executed by the processor without the programmers intervention.
- The programmer can write code that can “stall” the pipeline
- That will happen if the next instruction depends on the result of the previous instruction.

Example of a pipe stall

- Assume the following operations:

```
Instruction K:   C <= add A B
Instruction K+1: D <= sub E C
```

- The instruction K+1 needs the result of instruction K before it can continue.
- This causes instruction K+1 to wait until instruction k completes.

Example of a pipe stall

Clock	Stage1	Stage2	Stage3	Stage4	Stage5
1	Instk	instk-1	instk-2	instk-3	instk-4
2	Instk+1	Instk	instk-1	instk-2	instk-3
3	Instk+2	Instk+1	Instk	instk-1	instk-2
4	Instk+3	Instk+2	(Instk+1)	Instk	instk-1
5	-------	-------	(Instk+1)	------	Instk
6	-------	-------	Instk+1	------	-------
7	Instk+4	Instk+3	Instk+2	Instk+1	-------
8	Instk+5	Instk+4	Instk+3	Instk+2	Instk+1
9	Instk+6	Instk+5	Instk+4	Instk+3	Instk+2

Pipe Stall

- Some reasons of a pipe stall are:
  - Access to RAM
  - Call an instruction that takes a long time like multiplication, division, FP arithmetic
  - Branch to a new location
  - Call a function

Avoiding Pipe Stalls

- A programmer can delay the use of results by reordering the instructions:

Avoiding Stalls

- Program must be written to accommodate instruction pipeline
- To minimize stalls
  - – Avoid introducing unnecessary branches
  - – Delay references to result register(s)
  - - Avoid Memory access

Avoiding Stalls

- **Example Of Avoiding Stalls**
  - (a)
  - C ← add A B
  - D ← subtract E C
  - F ← add G H
  - J ← subtract I F
  - M ← add K L
  - P ← subtract M N

  (b)

  C ← add A B

  F ← add G H

  M ← add K L

  D ← subtract E C

  J ← subtract I F

  P ← subtract M N
- Stalls eliminated by rearranging (a) to (b)

Avoiding Stalls

- *Although hardware that uses an instruction pipeline will not run at full speed unless programs are written to accommodate the pipeline, a programmer can choose to ignore pipelining and assume the hardware will automatically increase speed whenever possible.*

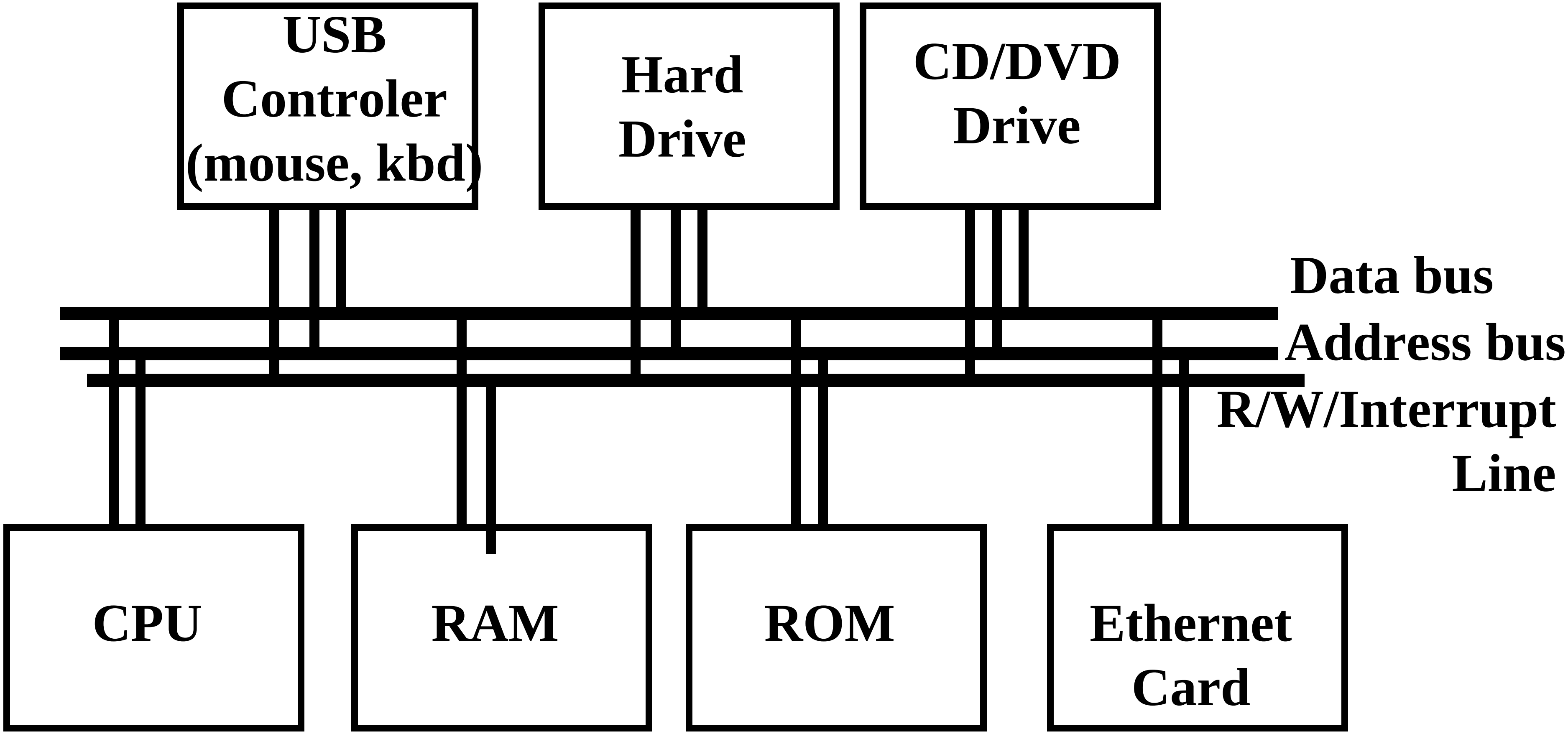
VII. CPUs Microcode Protection and Protection Modes

User and Kernel Mode, Interrupts, and System Calls

Computer Architecture Review

- Most modern computers use the ***Von Newman Architecture*** where both programs and data are stored in RAM.
- The alternative is the ***Harvard Architecture***, where programs and data are stored in separate memories.
- A computer has an ***address bus*** and a ***data bus*** that are used to transfer data from/to the CPU, RAM, ROM, and the devices.
- The CPU, RAM, ROM, and all devices are attached to this bus.

Computer Architecture Review



Kernel and User Mode

- Kernel Mode
  - When the CPU runs in this mode:
    - It can run any instruction in the CPU
    - It can modify any location in memory
    - It can access and modify any register in the CPU and any device.
    - There is full control of the computer.
  - The OS Services run in kernel mode.

Kernel and User Mode

- **User Mode**
  - When the CPU runs in this mode:
    - The CPU can only use a limited set of instructions
    - The CPU can only modify only the sections of memory assigned to the process running the program.
    - The CPU can access only a subset of registers in the CPU and it cannot access registers in devices.
    - There is a limited access to the resources of the computer.
  - The user programs run in user mode

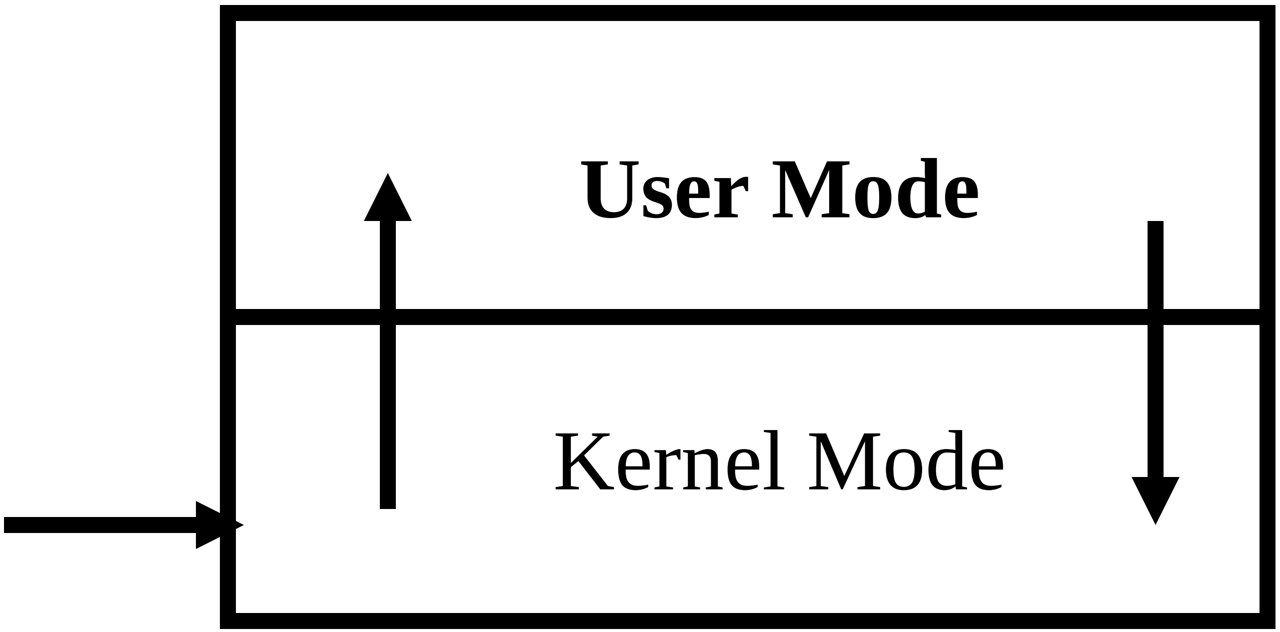
Kernel and User Mode

- When the OS boots, it starts in kernel mode.
- In kernel mode the OS sets up the interrupt vector and initializes all the devices, and kernel data structures.
- Then it starts the first process and switches to user mode.
- In user mode it runs all the background system processes (daemons).
- Then it runs the user shell or windows manager.

Kernel and User Mode

- User programs run in user mode.
- The programs switch to kernel mode to request OS services (system calls)
- Also user programs switch to kernel mode when an interrupt arrives.
- The interrupts are executed in kernel mode.
- The interrupt vector can be modified only in kernel mode.
- Most of the CPU time is spent in User mode

Kernel and User Mode



Kernel and User Mode

- Separation of user/kernel mode is used for:
  - Security: The OS calls in kernel mode make sure that the user has enough privileges to run that call.
  - Robustness: If a process in user mode tries to write to an invalid memory location, the OS will kill the process, but the OS continues to run. A crash in the process will not crash the OS. > A bug in user mode causes program to crash, OS runs. A bug in kernel mode may cause OS and system to crash.
  - Fairness: OS calls in kernel mode to enforce fair access.

Interrupts

- An interrupt is an event that requires immediate attention. In hardware, a device sets the interrupt line to high.
- When an interrupt is received, the CPU will stop whatever it is doing and it will jump to to the 'interrupt handler' that handles that specific interrupt.
- After executing the handler, it will return to the same place where the interrupt happened and the program continues. Examples:
  - move mouse
  - type key
  - ethernet packet

Steps of Servicing an Interrupt

1. The CPU saves the Program Counter and registers in execution stack
2. CPU looks up the corresponding interrupt handler in the interrupt vector.
3. CPU jumps to interrupt handler and run it.
4. CPU restores the registers and return back to the place in the program that was interrupted. The program continues execution as if nothing happened.
5. In some cases it retries the instruction that was interrupted (E.g. Virtual memory page fault handlers).

Running with Interrupts

- Interrupts allow CPU and device to run in parallel without waiting for each other.

1. OS Requests Device Operation (E.g.Write to disk)

2. Device Runs Operation

2. OS does other things in parallel with device.

3. When Operation is complete device interrupts OS

4. OS runs interrupt, returns and continues process

Polling

- Alternatively, the OS may decide not use interrupts for some devices and wait in a busy loop until completion.

```
OS requests Device operation
While request is not complete
   do nothing;
Continue execution.
```

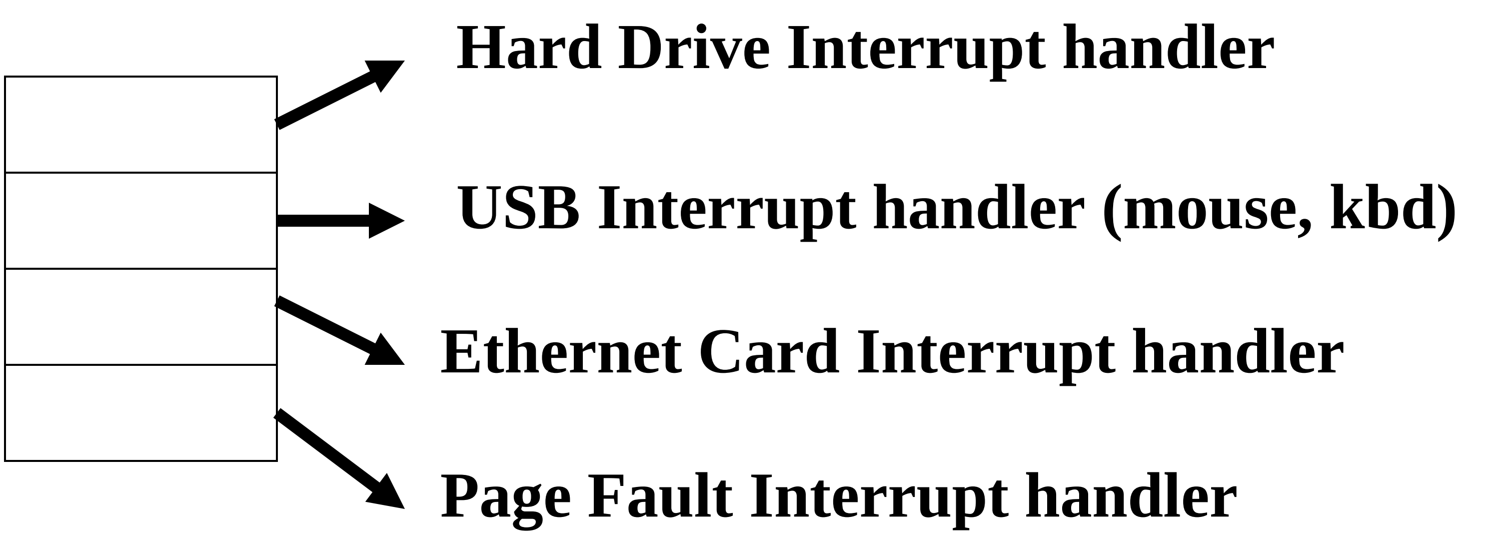
- This type of processing is called “polling” or “busy waiting” and wastes a lot of CPU cycles.
- Polling was used in early PCs when devices were simple and it is still in use in some microcontrollers.
- Polling is used for example to print debug messages in the kernel (kprintf). We want to make sure that the debug message is printed to before continuing the execution of the OS.

Synchronous vs. Asynchronous

- ***Polling*** is also called ***Synchronous Processing*** since the execution of the device is synchronized with the program.
- An interrupt is also called ***Asynchronous Processing*** because the execution of the device is not synchronized with the execution of the program. Both device and CPU run in parallel.

Interrupt Vector

- It is an array of pointer to functions that point to the different interrupt handlers of the different types of interrupts.



Interrupts and Kernel Mode

- Interrupts run in kernel mode. Why?
  - An interrupt handler must read device/CPU registers and execute instructions only available in kernel mode.
- Interrupt vector can be modified only in kernel mode (security)
- Interrupt vector initialized on bootup; modified when drivers added to system

Types of Interrupts

1. **Device Interrupts** generated by Devices when a request is complete or an event that requires CPU attention happens.
   - The mouse is moved
   - A key is typed
   - A WiFi/Ethernet packet arrives.
   - The hard drive/solid state drive has completed a read/write operation.
   - A CD has been inserted in the CD drive.

Types of Interrupts

2. **Math exceptions** generated by the CPU when there is a math error.
   - Divide by zero
3. ***Page Faults*** generated by the MMU (Memory Management Unit) that converts Virtual memory addresses to physical memory addresses
   - Invalid address: interrupt prompts a SEGV signal to the process
   - Page not resident. Access to a valid address but there is not page in memory. This causes the CPU to load the page from disk
   - Invalid permission (I.e. trying to write on a read only page) causes a SEGV signal to the process.

Types of Interrupts

4. **Software Interrupt** generated by software with a special assembly instruction (SYSCALL, INT, TRAP, TA). This is how a program running in user mode requests operating systems services.

System Calls

- System Calls is the way user programs request services from the OS
- **System calls** use **Software Interrupts**
- Examples of system calls are:
  - `int open(filename, mode)`
  - `read(file, buffer, size)`
  - `write(file, buffer, size)`
  - `int fork()`
  - `execve(cmd, args);`
- System calls is the API of the OS from the user program's point of view. See /usr/include/sys/syscall.h

Why do we use Software Interrupts for syscalls instead of function calls?

- Software Interrupts will switch into kernel mode
- OS services need to run in kernel mode because:
  - They need privileged instructions
  - Accessing devices and kernel data structures
  - They need to enforce the security in kernel mode.

System Calls

- Only operations that need to be executed by the OS in kernel mode are part of the system calls.
- Function like sin(x), cos(x) are not system calls.
- Some functions like printf(s) run mainly in user mode but eventually call ***`write()`*** when for example the buffer is full and needs to be flushed.
- Also ***malloc(size)*** will run mostly in user mode but eventually it will call ***sbrk()*** to extend the heap.

System Calls

- Libc (the C library) provides wrappers for the system calls that eventually generate the system calls.

User Mode:

```
int open(fname, mode) {
  return syscall(SYS_open,
  fname, mode);
}
int syscall(syscall_num, …)
{
  asm(INT);
  // or asm(syscall)
}
```

Software Interrupt

Kernel Mode:

```
Syscall interrupt handler:
switch (syscall_num){
caseSYS_read:…
caseSYS_write:…
caseSYS_open:
  - Get file name and mode
  - Verify file exists and
permissions of file against
mode.
  - Perform operation
  - return fd (file
```

System Calls

- The software interrupt handler for system calls has entries for all system calls.
- The handler checks that the arguments are valid and that the operation can be executed.
- The arguments of the syscall are checked to enforce the security and protections.

Syscall Security Enforcement

- For example, for the open syscall the following is checked in the syscall software interrupt handler:

  open(filename, mode)

  - If file does not exist return error
  - If permissions of file do not agree with the mode the file will be opened, return error. Consider also who the owner of the file is and the owner of the process calling open.
  - If all checks pass, open file and return file handler.

Syscall details

- The list of all system calls can be found in /usr/include/sys/syscall.h

```
#define SYS_exit        1
#define SYS_fork        2
#define SYS_read        3
#define SYS_write       4
#define SYS_open        5
#define SYS_close       6
#define SYS_wait        7
#define SYS_creat       8
#define SYS_link        9
#define SYS_unlink      10
#define SYS_exec        11
...
```

Syscall Error reporting

- When an error in a system call occurrs, the OS sets a global variable called “errno” defined in libc.so with the number of the error that gives the reason for failure.
- The list of all the errors can be found in
When
/usr/include/sys/errno.h

```
#define EPERM    1        /* Not super-user    */
#define ENOENT   2        /* No such file or directory */
#define ESRCH    3        /* No such process */
#define EINTR    4        /* interrupted system call  */
#define EIO      5        /* I/O error  */
#define ENXIO    6        /* No such device or address */
```

- You can print the corresponding error message to stderr using `perror(s);` where s is a string prepended to the message.

System Calls and Interrupts Example

1. The user program calls ***printf*** and then ***printf*** calls the **`write(fd, buff, n)`** system call to write to disk.
2. The write wrapper in libc generates a software interrupt for the system call.
3. The OS in the interrupt handler checks the arguments. It verifies that fd is a file descriptor for a file opened in write mode. And also that [buff, buff+n] is a valid memory range. If any of the checks fail write return -1 and sets errno to the error value.

System Calls and Interrupts Example

4. The OS tells the hard drive to write the buffer in [buff, buff+n] to disk to the file specified by fd.
5. The OS puts the current process in wait state until the disk operation is complete. Meanwhile, the OS switches to another process.
6. The Disk completes the write operation and generates an interrupt.
7. The interrupt handler puts the process calling `write` into ready state so this process will be scheduled by the OS in the next chance.

Using syscalls

```
#include <stdio.h>
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <unistd.h>
#include <sys/syscall.h>   /* For SYS_xxx definitions */

int counter = 0;

int
main()
{
        printf("Hello cs252 with printf\n");
        write(1, "Hello cs252 with write\n", 23);
        syscall(SYS_write, 1, "Hello cs252 with syscall\n", 25);
}
```

Syscall in assembly X86-64

```
syscall-write.s

.text
message:
        .string  "Hello, World\n"
.global main
main:
        # write(1, message, 13)
        movq     $1, %rax                   # system call 1 is write
        movq     $1, %rdi                   # file handle 1 is stdout
        movq     $message,%rsi              # address of string to output
        movq     $13, %rdx                  # number of bytes
        syscall

        # exit(0)
        movq     $60, %rax                # system call 60 is exit
        movq     $0, %rdi                 # return code 0
        syscall
```

Syscall in assembly X86-64

```
grr@data:$ gcc -g -static -o syscall-write syscall-write.s

grr@data:$ ./syscall-write

Hello, World
```

Virtual Memory

Virtual Memory Introduction

- VM was created to allow running processes with memory requirements larger than available RAM to run in the computer.
- If the following processes are running with the noted requirements:
  - Chrome (100MB),
  - MSWord (100MB),
  - Yahoo Messenger (30MB)
  - Operating System (200MB).
- This would require 430MB of memory when there may only be 256MB of RAM available
- Currently, VM is used mainly to give each process its own address space.

Virtual Memory Introduction

- VM only keeps in RAM the memory that is currently in use.
- The remaining memory is kept in disk in a special file called "swap space"
- The VM idea was created by Peter Denning a former head of the CS Department at Purdue

Other Uses of Virtual Memory

Other uses of VM are:

-Give each process its own address space.

E.g. Address 0x10000 in one process p1 will use different physical memory than address 0x10000 in another process p2.

- Speed up some of the tasks in the OS for example:
  - Loading a program. The VM will load pages of the program as they are needed, instead of loading the program all at once.
  - During fork the child gets a copy of the memory of the parent. However, parent and child will use the same memory as long as it is not modified, making the fork call faster. This is called “copy-on-write”.
  - Shared Libraries across processes.
  - Shared memory
  - There are other examples that we will cover later.

VM Implementations

- Process Swapping:
  - The entire memory of the process is swapped in and out of memory
- Segment Swapping
  - Entire parts of the program (process) are swapped in and out of memory (libraries, text, data, bss, etc.

  - Problems of process swapping and segment swapping is that the granularity was too big and some pieces still in use could be swapped out together with the pieces that were not in use.
- Paging
  - Used by modern OSs. Covered in detail here.

Paging

- Implementation of VM used by modern operating systems.
- The unit of memory that is swapped in and out is a page
- Paging divides the memory in pages of fixed size.
- Usually the size of a page is 4KB in the Pentium (x86) and ARM architecture and 8KB in the Sparc Ultra Architecture/Alpha.

Paging

Address in bytes		VM Address in pages (page numbers)
0xFFFFFFFF $2^{32}-1=4G-1$		$2^{32}/4KB-1 = 2^{20}-1=2M-1$
	Not mapped(invalid)	
•	Swap page 500	
	RAM page 3	
•	RAM page 10	
	Swap page 456	
•	RAM page 5	
	Executable page 2	2
0x00002000 8192	RAM page 24	1
0x00001000 4096		0
0x00000000 0		

Paging

- The Virtual Memory system will keep in memory the pages that are currently in use.
- It will leave in disk the memory that is not in use.

Backing Store

- Every page in the address space is backed by a file in disk, called ***backing-store***

Memory Section	**Permissions (RWX)**	***Backing Store***
Text	R-X	Executable File
ROData (Read Only Data)	R--	Executable File
Data	R-- – At the beginning RW- – After modified	Executable File when page is not not modified. Swap space when page is modified
BSS	RW-	Swap Space
Stack	RW-	Swap Space
Heap	RW-	Swap Space

Swap Space

- Swap space is a designated area in disk that is used by the VM system to store transient data.
- In general any section in memory that is not persistent and will go away when the process exits is stored in swap space.
- Examples: Stack, Heap, BSS, modified data etc.

Swap Space

```
lore 208 $ df -k
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/dsk/c0t0d0s0    1032130  275238  705286    29%    /
/proc                      0       0       0     0%    /proc
mnttab                     0       0       0     0%    /etc/mnttab
fd                         0       0       0     0%    /dev/fd
/dev/dsk/c0t0d0s4    2064277 1402102  600247    71%    /var
swap                  204800    2544  202256     2%    /tmp
/dev/dsk/c0t2d0s6    15493995 11682398 3656658    77%    /.lore/u92
/dev/dsk/c0t3d0s6    12386458 10850090 1412504    89%    /.lore/u96
/dev/dsk/c0t1d0s7    15483618 11855548 3473234    78%    /.lore/u97
bors-2:/p8           12387148 8149611 4113666    67%    /.bors-2/p8
bors-2:/p4           20647693 11001139 9440078    54%    /.bors-2/p4
xinuserver:/u3       8744805 7433481 1223876    86%    /.xinuserver/u3
galt:/home           5161990 2739404 2370967    54%    /.galt/home
xinuserver:/u57      15481270 4581987 10775435    30%    /.xinuserver/u57
lucan:/p24           3024579 2317975  676359    78%    /.lucan/p24
ector:/pnews         8263373  359181 7821559     5%    /.ector/pnews
```

Swap Space

```
lore 206 $ /usr/sbin/swap -s
total: 971192k bytes allocated + 1851648k reserved = 2822840k used, 2063640k available

lore 207 $ /usr/sbin/swap -l
swapfile             dev  swaplo blocks   free
/dev/dsk/c0t0d0s1   32,1025     16 2097392 1993280
/dev/dsk/c0t1d0s1   32,1033     16 2097392 2001792
```

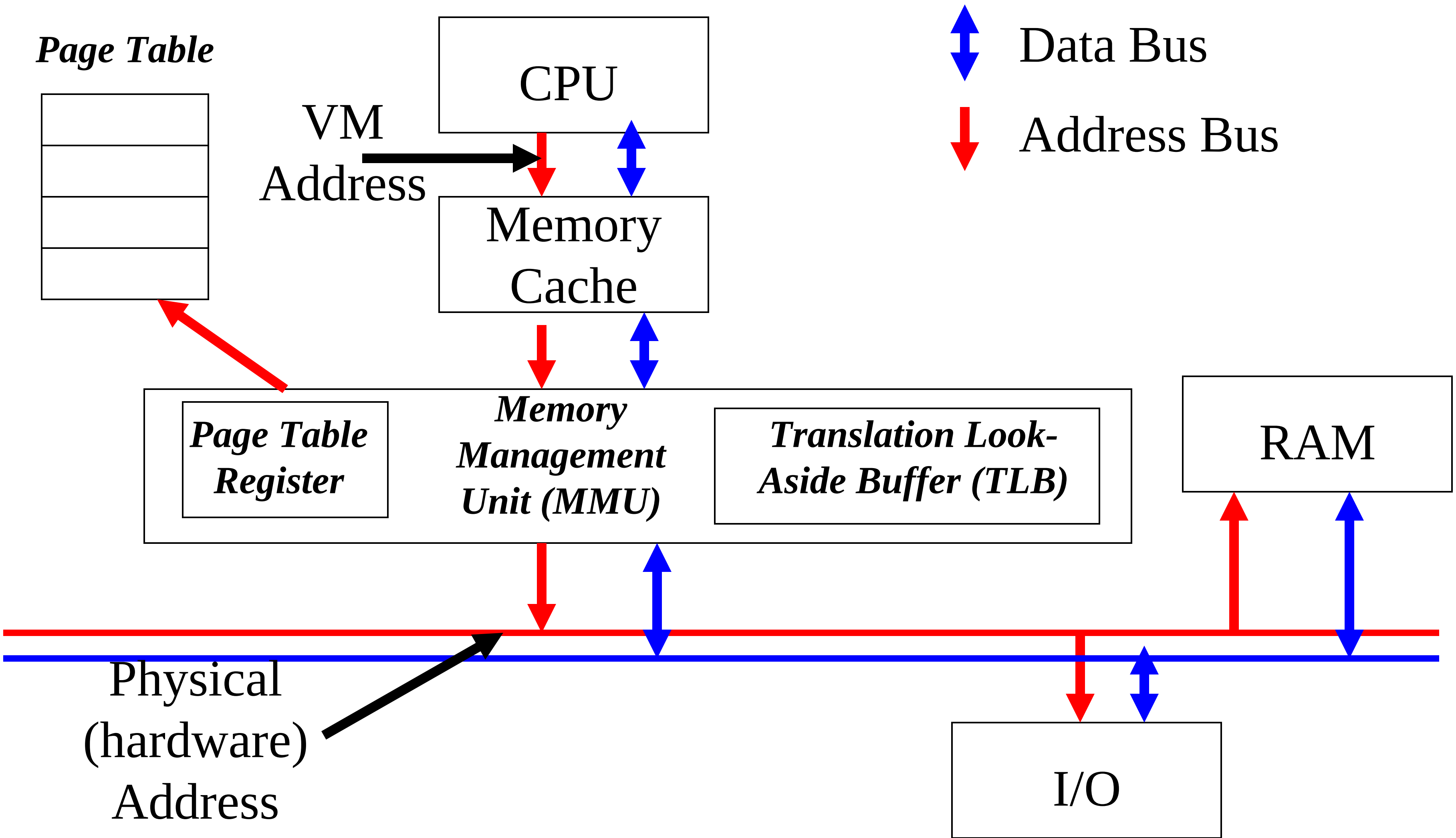
Implementation of Paging

- Paging adds an extra indirection to memory access.
- This indirection is implemented in hardware, so it does not have excessive execution overhead.
- The Memory Management Unit (MMU) translates Virtual Memory Addresses (vmaddr) to physical memory addresses (phaddr).
- The MMU uses a page table to do this translation.

Paging

- There are two types of addresses:
  - ***Virtual Memory Addresses***: the address that the CPU is using. Addresses used by programs are of this type.
  - ***Physical Memory Addresses***: The addresses of RAM pages. This is the **hardware address**.
- The MMU translates the Virtual memory addresses to physical memory addresses

The Memory Management Unit



The Memory Management Unit

- The MMU has a Page Table Register that points to the current page table that will be used for the translation.
- Each process has a its own page table.
- The page table register is updated during a context switch from one process to the other.
- The page table has the information of the memory ranges that are valid in a process

The Memory Management Unit

- The value of the ***page table register*** changes every time there is a context switch from one process to another.
- Consecutive pages in Virtual memory may correspond to non-consecutive pages in physical memory.

The Memory Management Unit

- To prevent looking up the page table at every memory access, the most recent translations are stored in the Translation Look-Aside buffer (TLB).
- The TLB speeds up the translation from virtual to physical memory addresses.
- A page fault is an interrupt generated by the MMU

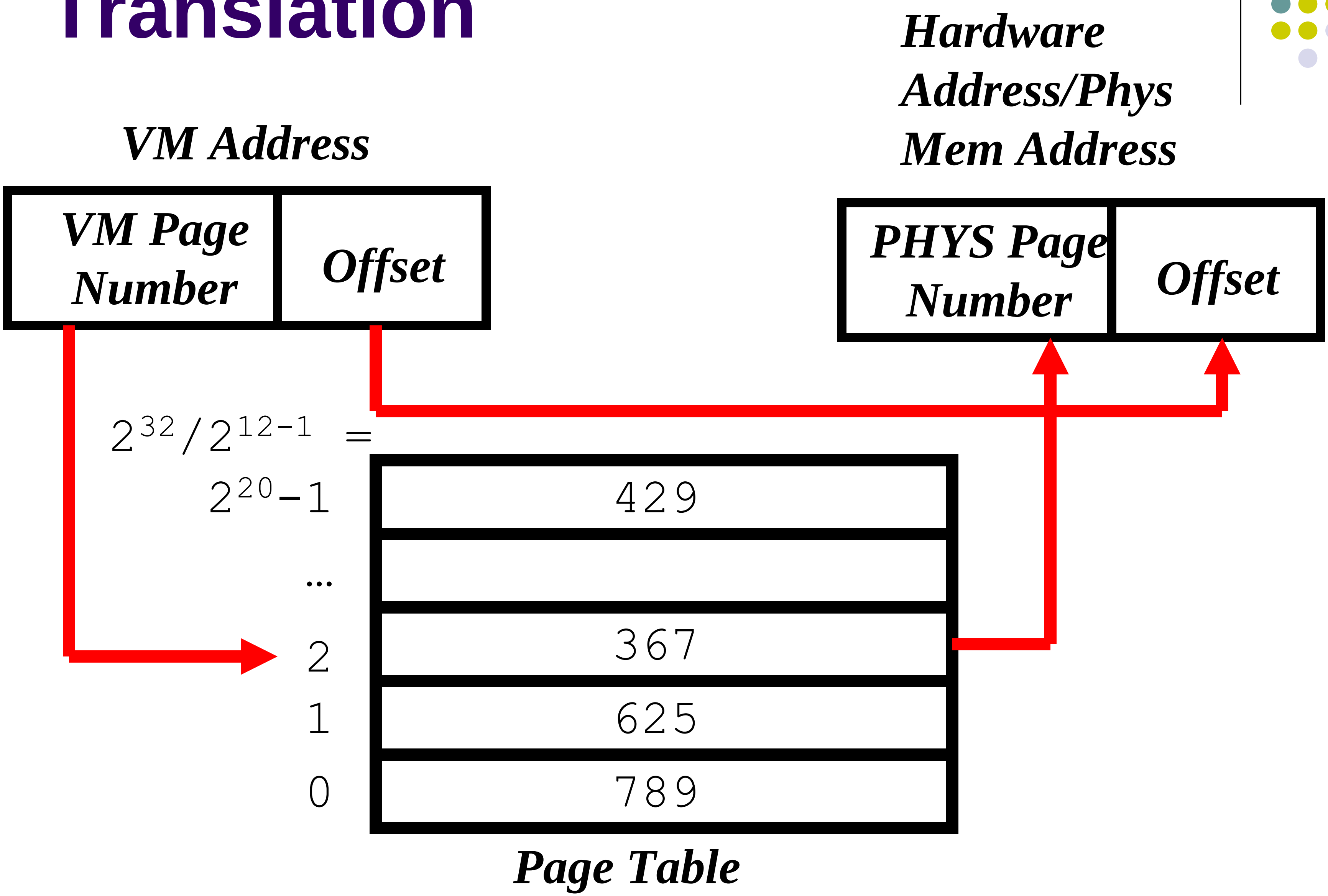
VM to Hardware Address Translation

- The VM address is divided into two parts:
  - Page number (higher 20 bits)
  - Offset (Lower 12 bits: 0-4095) (Assuming page size=4096 or 2^{12})

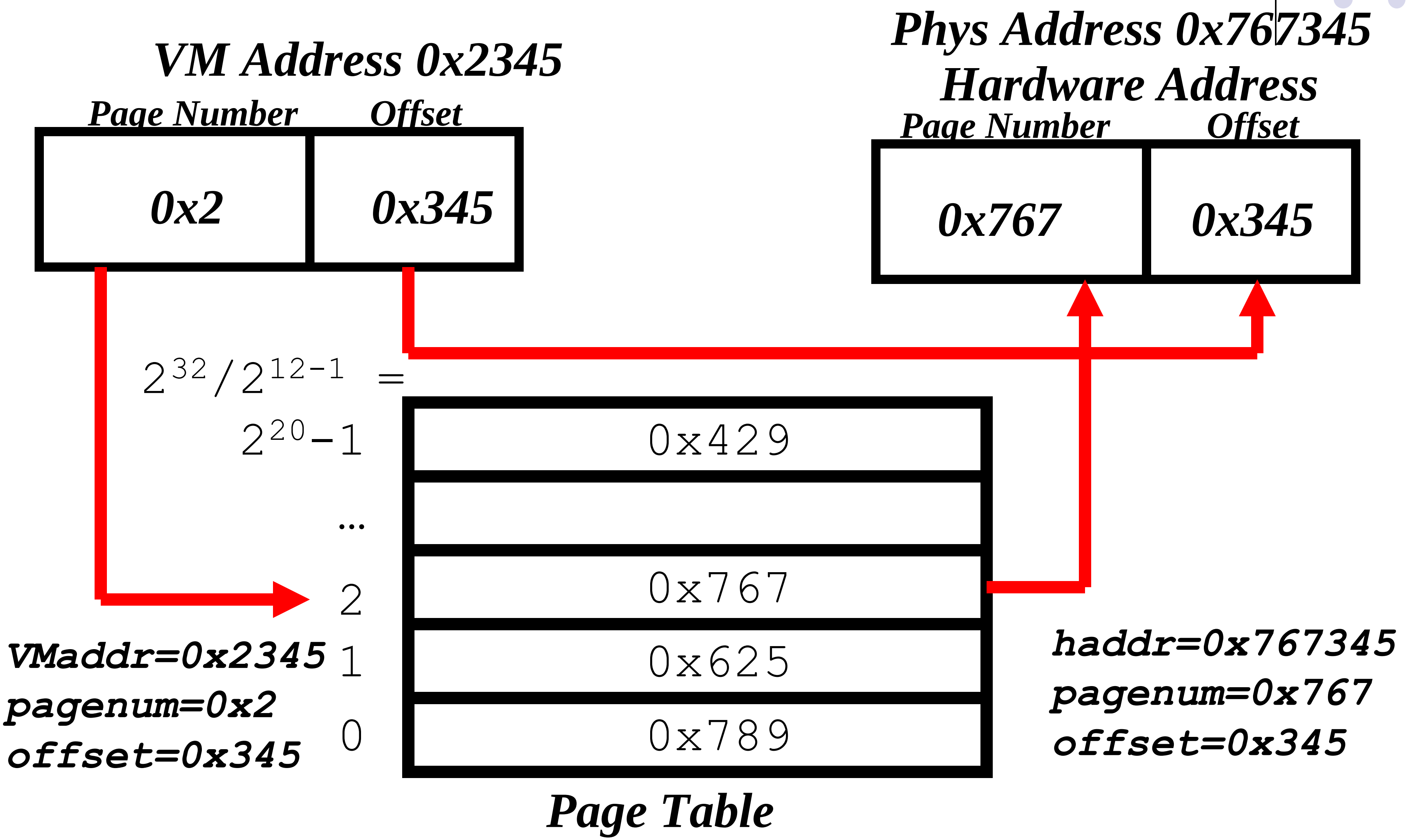
Page number	***Offset***
31 — 12	11 — 0

- Only the page number is translated. The offset remains the same
- Example: in 0x2345, the last 3 hex digits (12 bits) is the offset: 0x345. The remaining digits is the page number (20 bits): 0x2

VM to Hardware Address Translation



VM to Hardware Address Translation (one-level page table)



Two-Level page tables

- Using a one-level page table requires too much space: 2^{20} entries * 4 bytes/entry =~ 4MB.
- Since the virtual memory address has a lot of gaps, most of these entries will be unused.
- Modern architectures use a multi-level page table to reduce the space needed

Two-Level page tables

- The page number is divided into two parts: first-level page number and the second-level page number

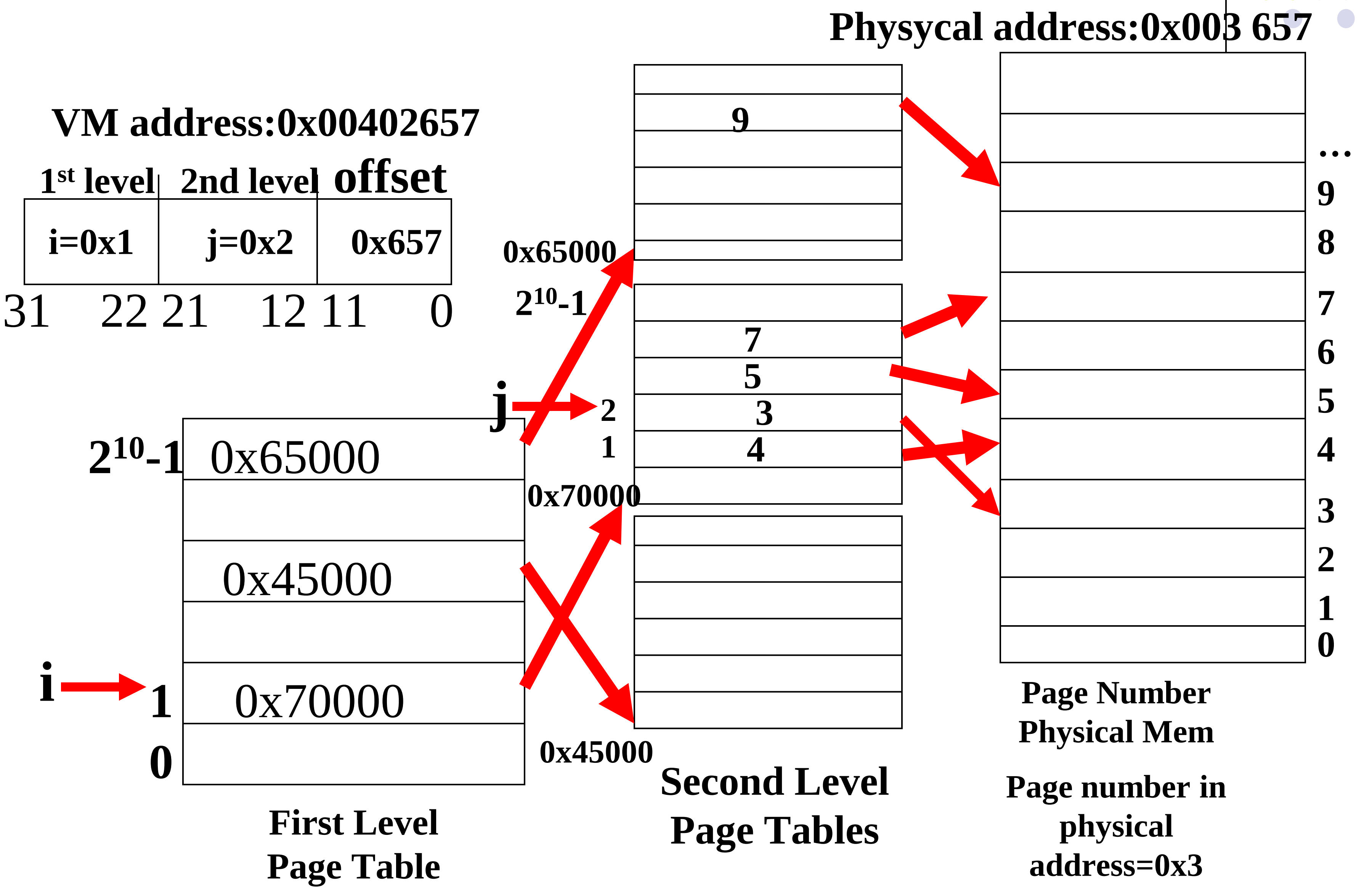
First-level index (i) (10 bits)	Second-level index (j) (10 bits)	Offset (12 bits)

- VMAddress = 0x00402657

0000 0000 01	00 0000 0010	0110 0101 0111
First level	Second level	Offset

- Offset=0x657 (last 3 hex digits)
- 1st level index (i) = 0x1 , 2nd level index (j)= 0x2

VM Address Translation



Example

- VMaddress: 0x00402 657
- Physical Memory Address: 0x3 657

  1.From the VM address find i, j, offset

  2. SecondLevelPageTable= FirstLevelPageTable[i]
  3. PhysMemPageNumber = SecondLevelPageTable[j]
  4. PhysMemAddr= PhysMemPageNum*Pagesize + offset
- Processes always have a first-level page table
- Second level page tables are allocated as needed.
- Both the first level and second level page tables use 4KB.

Page Bits

- Each entry in the page table needs only 20 bits to store the page number. The remaining 12 bits are used to store characteristics of the page.
  - Resident Bit:
    - Page is resident in RAM instead of swap space/file.
  - Modified Bit:

    Page has been modified since the last time the bit was cleared. Set by the MMU.
  - Access Bit:
    - Page has been read since the last time the bit was cleared. Set by MMU
  - Permission:

    Read page is readable

    Write Page is writable

    Execute Page can be executed (MMU enforces permissions)

Page Bits

- If a CPU operation exceeds the permissions of a page, like writing on a read-only page, the MMU will generate an interrupt (page fault). The interrupt may be translated into a signal (SEGV, SIGBUS) to the process.
- If a page is accessed and the page is not resident in RAM, the MMU generates an interrupt to the kernel and the kernel loads that page from disk.

Types of Page Fault

- A Page Fault is an interrupt generated by the MMU
- Page Fault
  - **Page not Resident**: Page not in Physical Memory, it is in disk
  - **Protection Violation**: Write, Read/Execute permissions (as indicated by page bits) are violated.

Processing a Page Fault

1. A program tries to read/write a location in memory that is in a non-resident page. This could happen when:
   - fetching the next instruction to execute or
   - trying to read/write memory not resident in RAM
2. The MMU tries to look up the VM address and finds that the page is not resident using the resident bit. Then the MMU generates a page fault, that is an interrupt from the MMU
3. Save return address and registers in the stack

Processing a Page Fault

4. The CPU looks up the interrupt handler that corresponds to the page fault in the interrupt vector and jumps to this interrupt handler
5. In the page fault handler

   If the VM address corresponds to a page that is not valid for this process, then generate a SEGV signal to the process. The default behavior for SEGV is to kill the process and dump core

   Otherwise, if VM address is in a valid page, then the page has to be loaded from disk.

Processing a Page Fault

6. Find a free page in physical memory. If there are no free pages, then use one that has not been used recently and write to disk if modified
7. Load the page from disk and update the page table with the address of the page replaced. Also, clear the modified and access bits. Set resident bit to 1.
8. Restore registers, return and retry the offending instruction.

Processing a Page Fault

- The page fault handler retries the offending instruction at the end of the page fault
- The page fault is completely transparent to the program, that is, the program will have no knowledge that the page fault occurred.

Using mmap

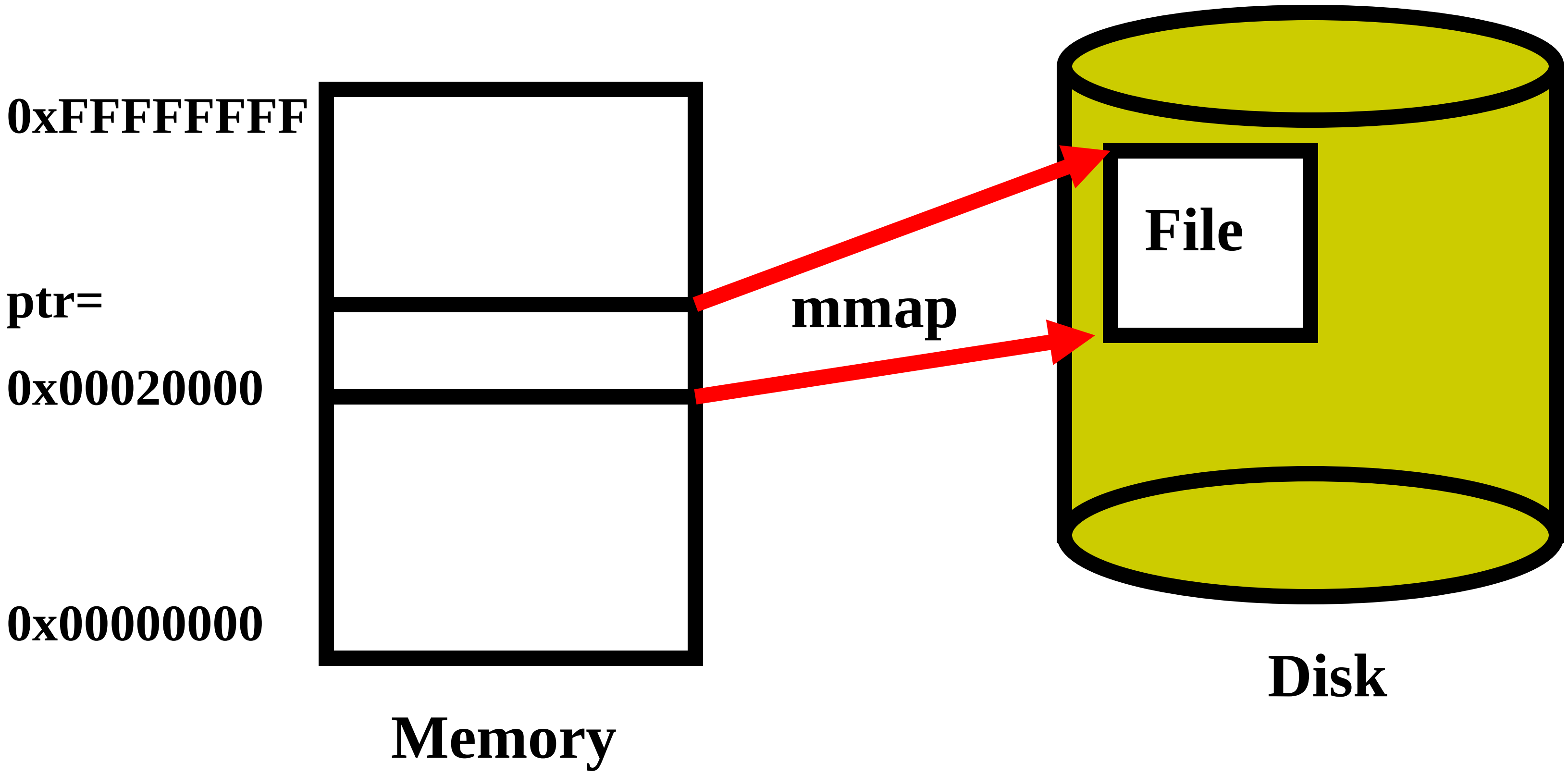
- The mmap() function establishes a mapping between a process's address space and a file or shared memory object.

```
#include <sys/mman.h>
void *mmap(void *addr, size_t len, int prot,
            int flags, int fildes, off_t off);
```

- Mmap returns the address of the memory mapping and it will be always aligned to a page size (addr%PageSize==0).
- The data in the file can be read/written as if it were memory.

Using mmap

ptr = mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0)



Mmap parameters

```
void *mmap(void *addr, size_t len, int prot,
              int flags, int fildes, off_t off);
```

addr –

Suggested address. If NULL is passed the OS will choose the address of the mapping.

len –

Length of the memory mapping. The mmaped file should have this length or larger or the program gets SEGV on access.

Prot –

Protections of the mapping: PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE. –

Mmap parameters

flags: - Semantics of the mapping:

- MAP_SHARED – Changes in memory will be done in the file
- MAP_PRIVATE – Changes in memory will be kept private to the process and will not be reflected in the file. This is called "copy-on-write"
- MAP_FIXED – Force to use "addr" as is without changing. You should know what you are doing since the memory may be already in use. Used by loaders
- MAP_NORESERVE– Do not reserve swap space in advance. Allocate swap space as needed.
- MAP_ANON – Anonimous mapping. Do not use any fd (file). Use swap as the backing store. This option is used to allocate memory

Fd –

The file descriptor of the file that will be memory mapped. You need to call open() to open the file and get the fd. Pass –1 if MAP_ANON is used.

Offset –

Offset in the file where the mapping will start. It has to be a multiple of a page size.

Mmap returns MAP_FAILED ((void*)-1) if there is a failure.

Notes on mmap

- Writing in memory of a memory-mapped file will also update the file in the disk (Only with MAP_SHARED).
- Updating the disk will not happen immediately.
- The OS will "cache" the change until it is necessary to flush the changes.
  - When the file is closed
  - Periodically (every 30secs or so)
  - When the command "sync" is typed
- If you try to read the value from the file of a page that has not been flushed to disk, the OS will give you the most recent value from the memory instead of the disk.

Example of using mmap

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/mman.h>
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>

// Show how to uyse mmap by writing hello to a file using memory
int main(int argc, char **argv)
{
        // Create file
        int fd = open( "hello.txt", O_RDWR|O_CREAT, 0770);
        if ( fd < 0) {
                perror("open");
                exit(1);
        }

        // Write 4096 zeroes into the file by seeking at that location and writing
        lseek(fd, 4096, SEEK_SET);
        write(fd, "", 1);
```

Example of using mmap

```
        // Create memory mapping
        char * addr = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                          MAP_SHARED, fd, 0);

        close(fd);

        if (addr == MAP_FAILED) {
                perror("mmap");
                exit(1);
        }

        strcpy(addr, "Hello world\n");

}
```

```
grr@data:~/cs250/Summer2020$ rm hello.txt
grr@data:~/cs250/Summer2020$ gcc -o mmap mmap.c
grr@data:~/cs250/Summer2020$ ./mmap
grr@data:~/cs250/Summer2020$ cat hello.txt
Hello world
```

Uses of VM

- The VM is not only to be able to run programs that use more memory than the RAM available.
- The main use of VM is to give each process a separate Virtual Memory Address space.
- VM also speeds up the execution of programs:
  1. Mmap the text segment of an executable or shared library
  2. Mmap the data segment of a program
  3. Use of VM during fork to copy memory of the parent into the child
  4. Allocate zero-initialized memory. it is used to allocate space for bss, stack and sbrk()
  5. Shared Memory

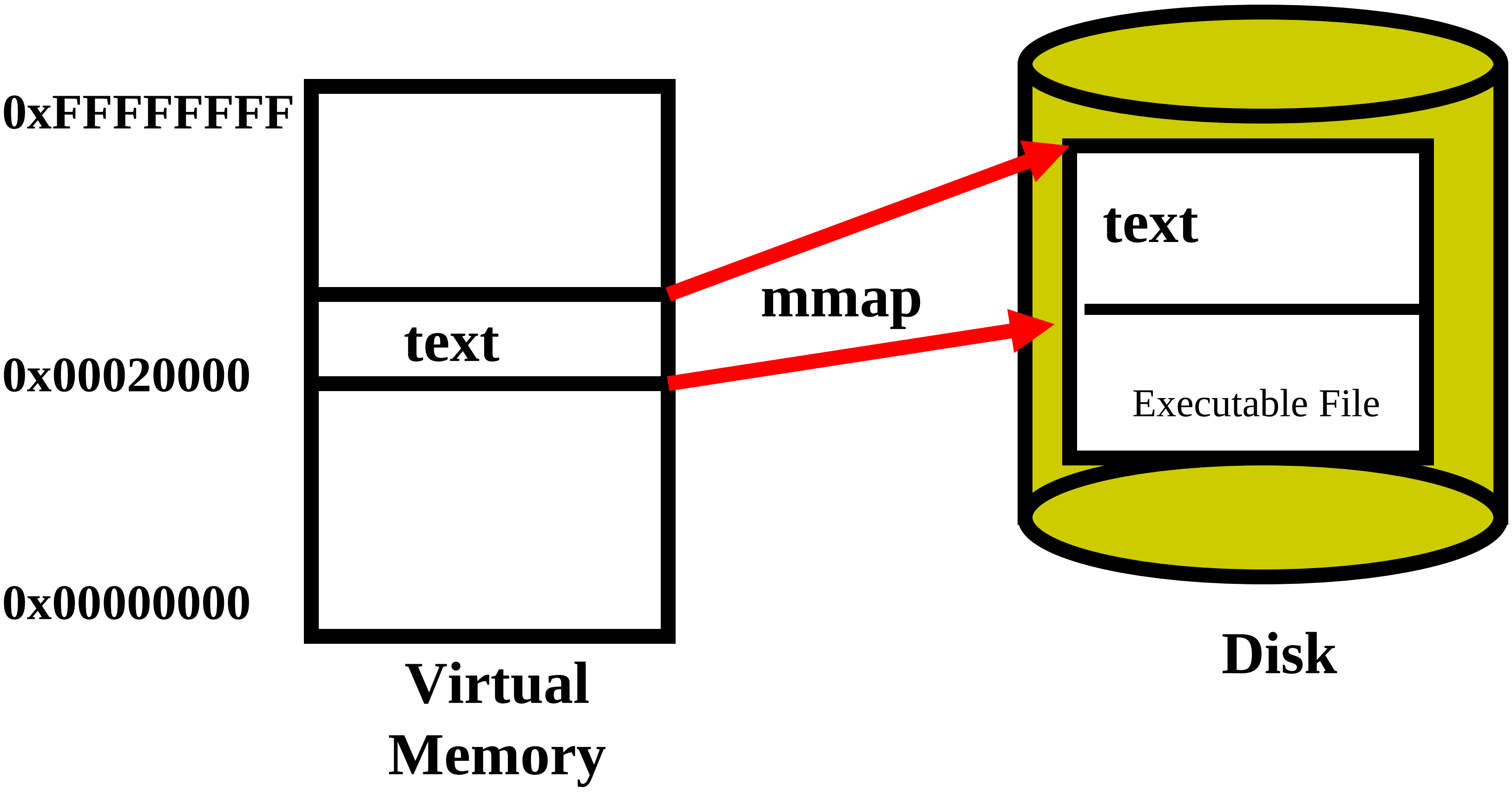
1. Mmap the text segment of an executable or a shared library

- initially mmap does not read any pages
- any pages will be loaded on demand when they are accessed
- startup time is fast because only the pages needed will be loaded instead of the entire program
- It also saves RAM because only the portions of the program that are needed will be in RAM

1. Mmap the text segment of an executable or a shared library

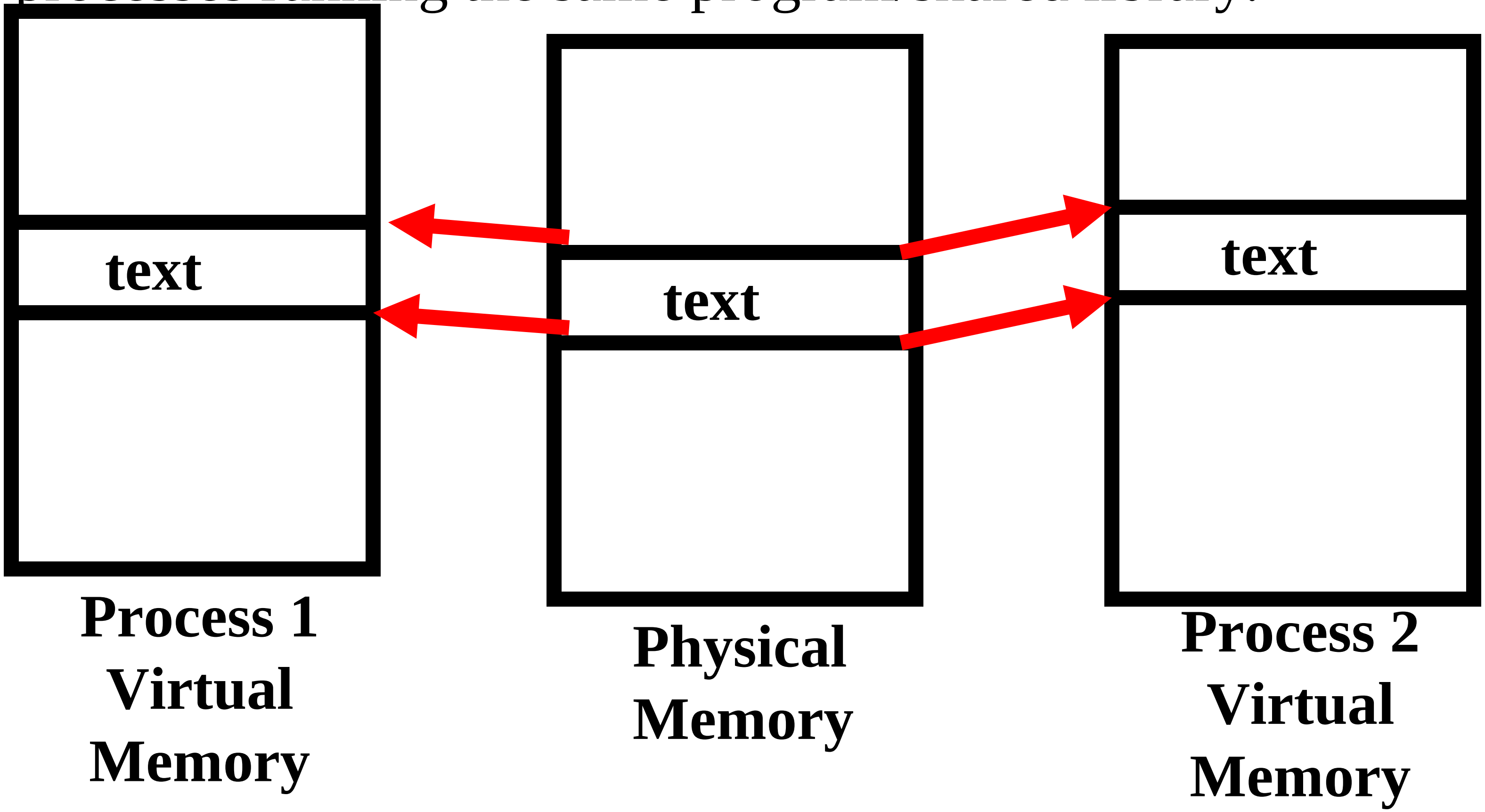
- Physical pages where the text segment is stored is shared by multiple instances of the same program.
- Protections: PROT_READ|PROT_EXEC
- Flags: MAP_PRIVATE

1. Mmap the text segment of an executable or a shared library



1. Mmap the text segment of an executable or a shared library

Physical Pages of the text section are shared across multiple processes running the same program/shared library.



Displaying the elf format of an executable

% readelf -a hello | less

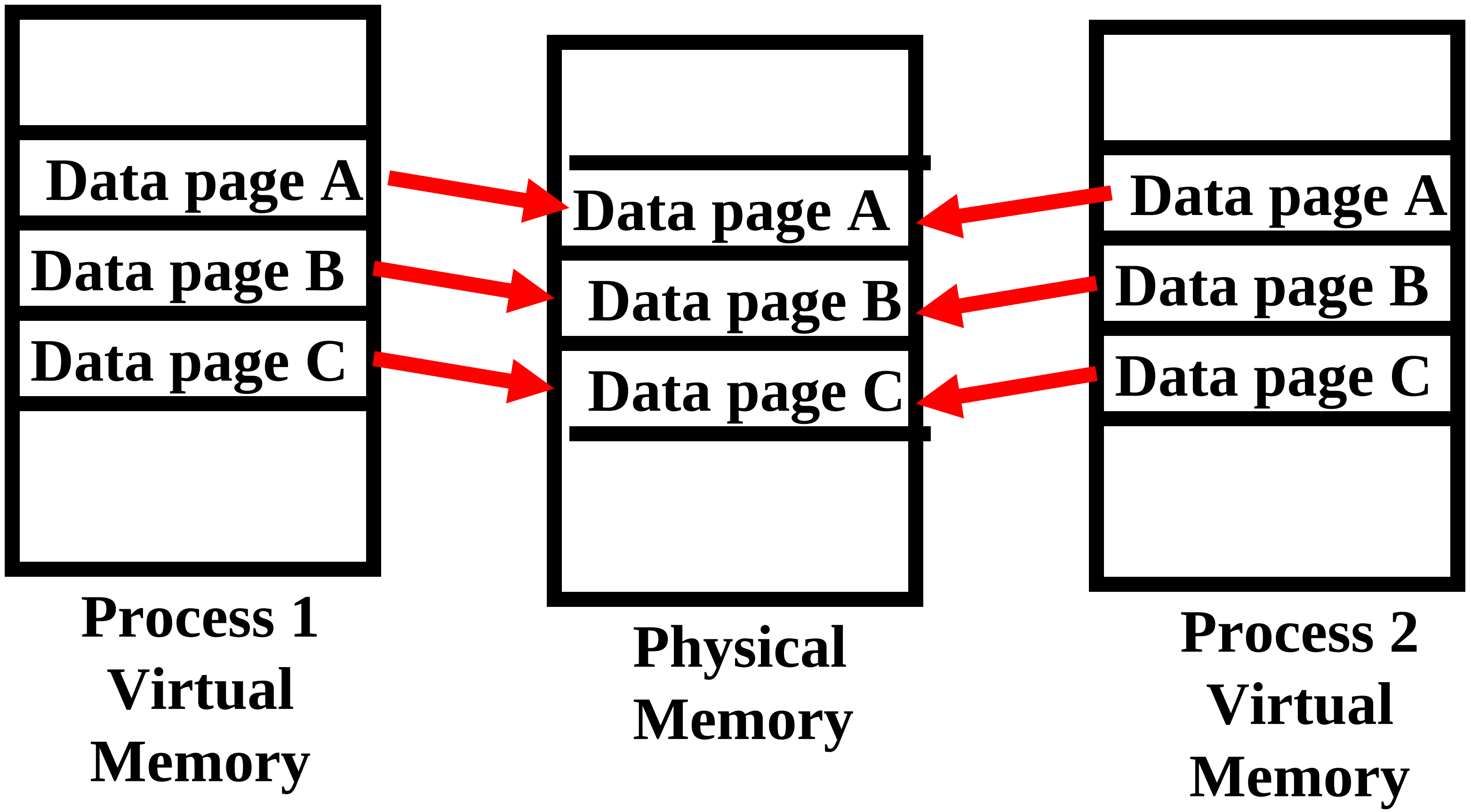
```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x5d0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          8784 (bytes into file)
  Flags:                             0x0
```

2. Mmap the data segment of a program

- During the loading of a program, the OS mmaps the data segment of the program
- The data segment contains initialized global variables.
- Multiple instances of the same program will share the same physical memory pages where the data segment is mapped as long as the page is not modified
- If a page is modified, the OS will create a copy of the page and make the change in the copy. This is called "copy on write"

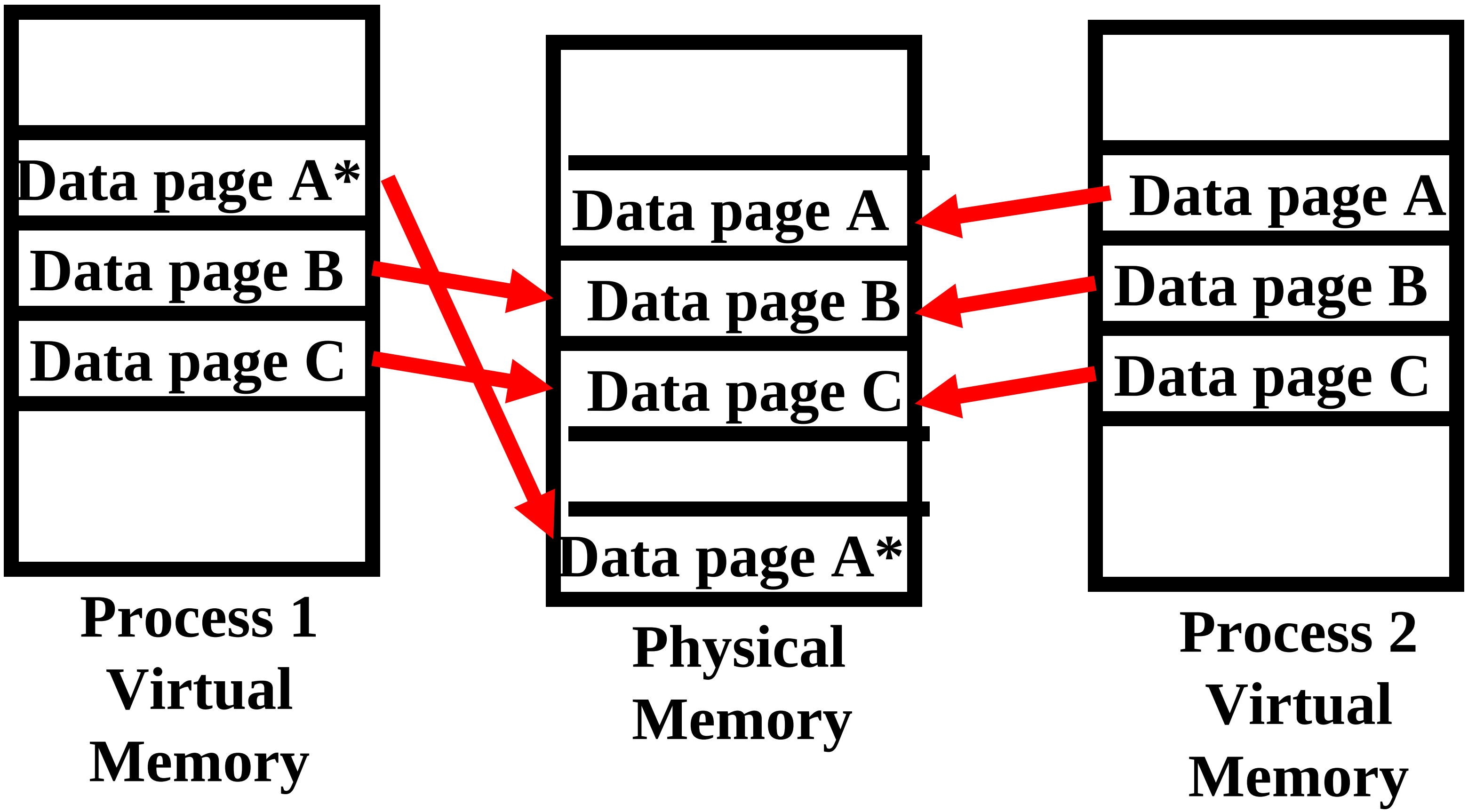
2. Mmap the data segment of a program

- Processes running the same program will share the same unmodified physical pages of the data segment



2. Mmap the data segment of a program

- When a process modifies a page, it creates a private copy (A*). This is called copy-on-write.



Using fork()

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

int counter = 2;

int
main()
{
        int ret = fork();
        printf("After fork ret=%d\n", ret);
        if (ret ==0) {
                printf("I am the child process counter=%d\n", counter);
                counter = 7;
        }
        else {
                 printf("I am the parent process counter=%d\n",counter);
                counter = 25;
        }
        printf("ret=%d counter=%d\n", ret, counter);
}
```

3. Use of VM during fork to copy memory of the parent into the child

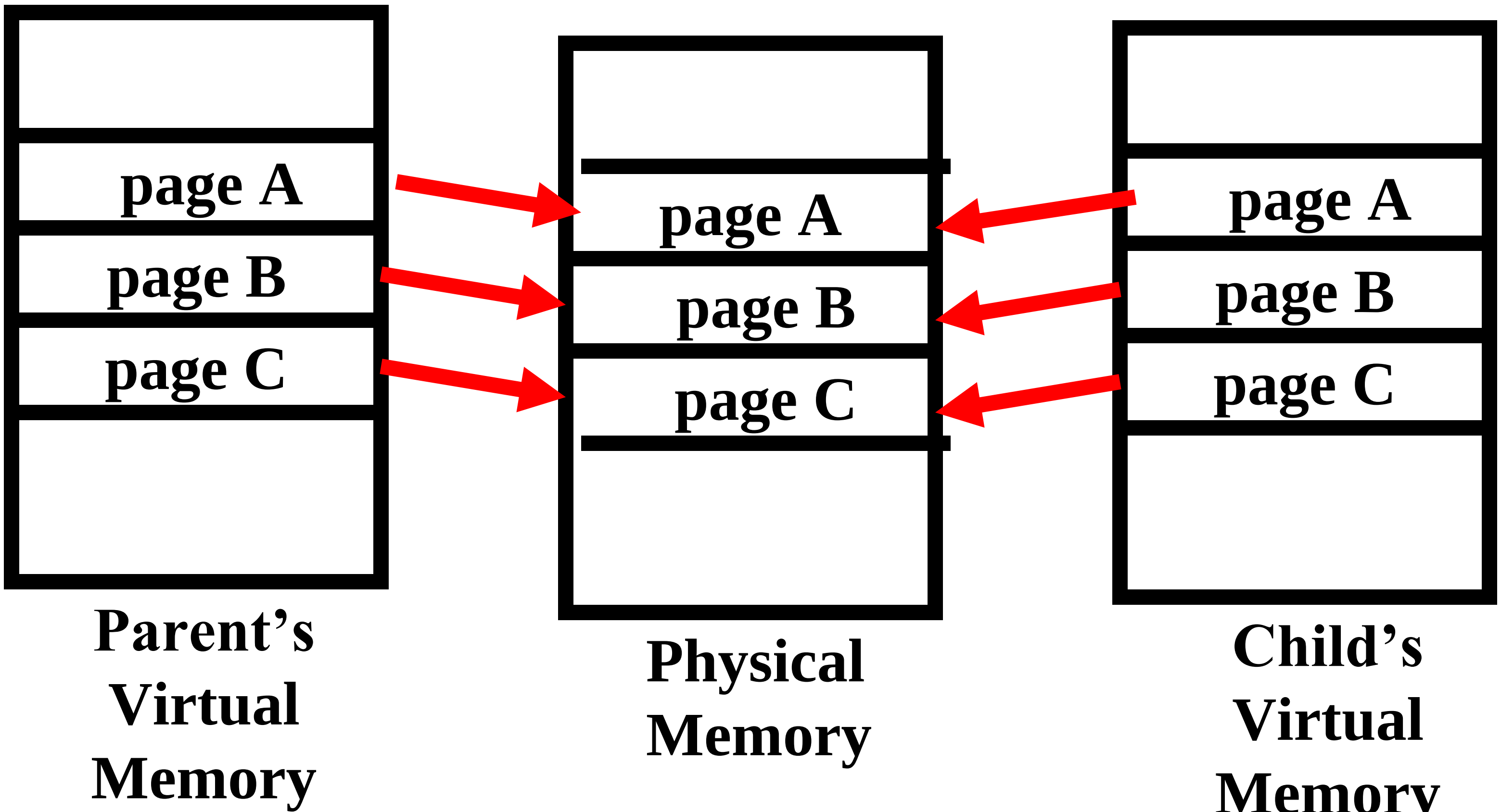
- After forking, the child gets a copy of the memory of the parent
- The page table of the parent is copied into the child page table, and both parent and child share the same RAM pages (physical memory) as long as they are not modified
- When a page is modified by either parent or child, the OS will create a copy of the page in RAM and will do the modifications on the copy

3. Use of VM during fork to copy memory of the parent into the child

- The copy on write in fork is accomplished by making the common pages read-only.
- The OS will catch the modifications. During the page fault will create a copy and update the page table of the writing process.
- Then it will retry the modify instruction.

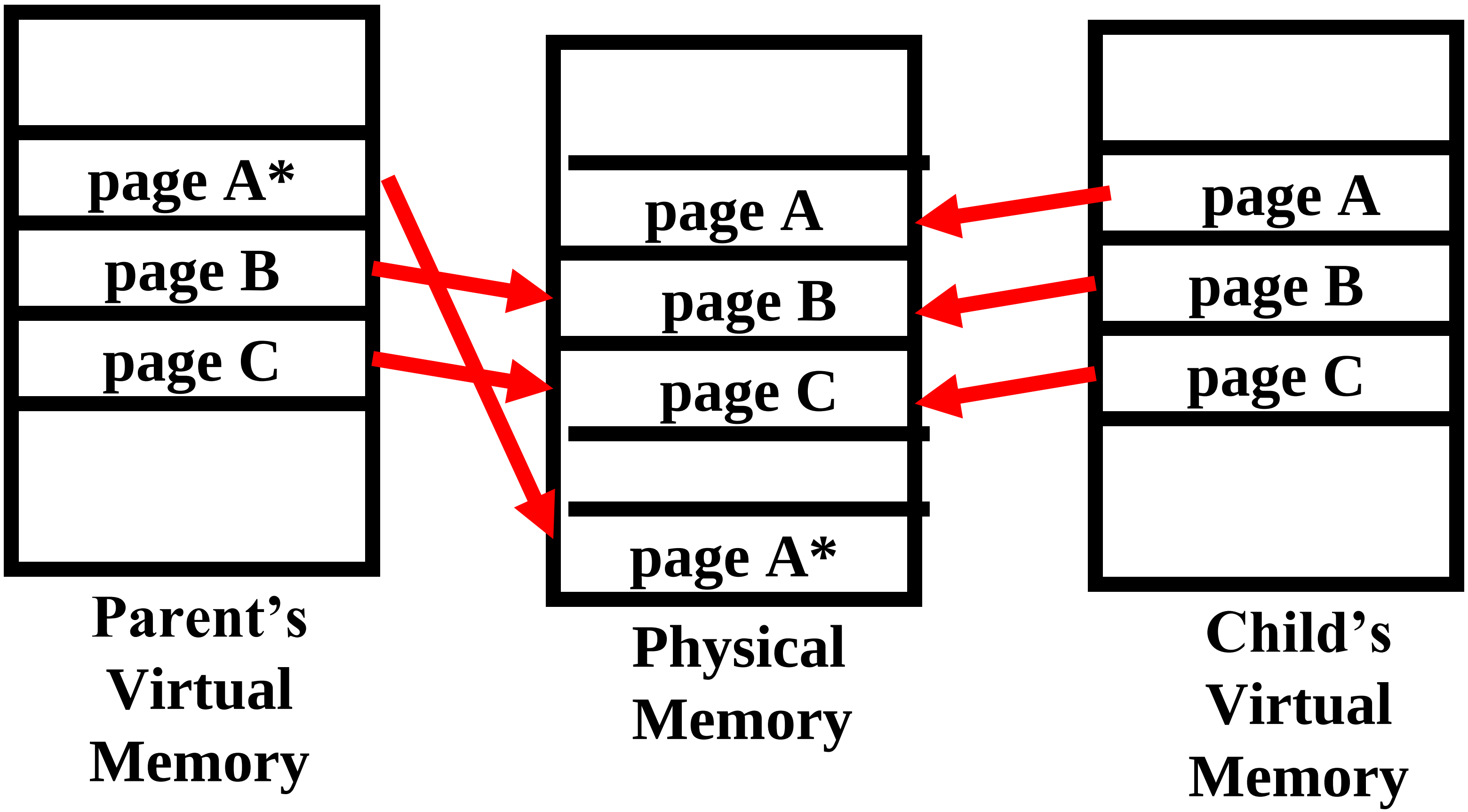
3. Use of VM during fork to copy memory of the parent into the child

- After fork() both parent and child will use the same pages



3. Use of VM during fork to copy memory of the parent into the child

- When the child or parent modifies a page, the OS creates a private copy (A*) for the process. This is called copy-on-write.



4. Allocate zero-initialized memory.

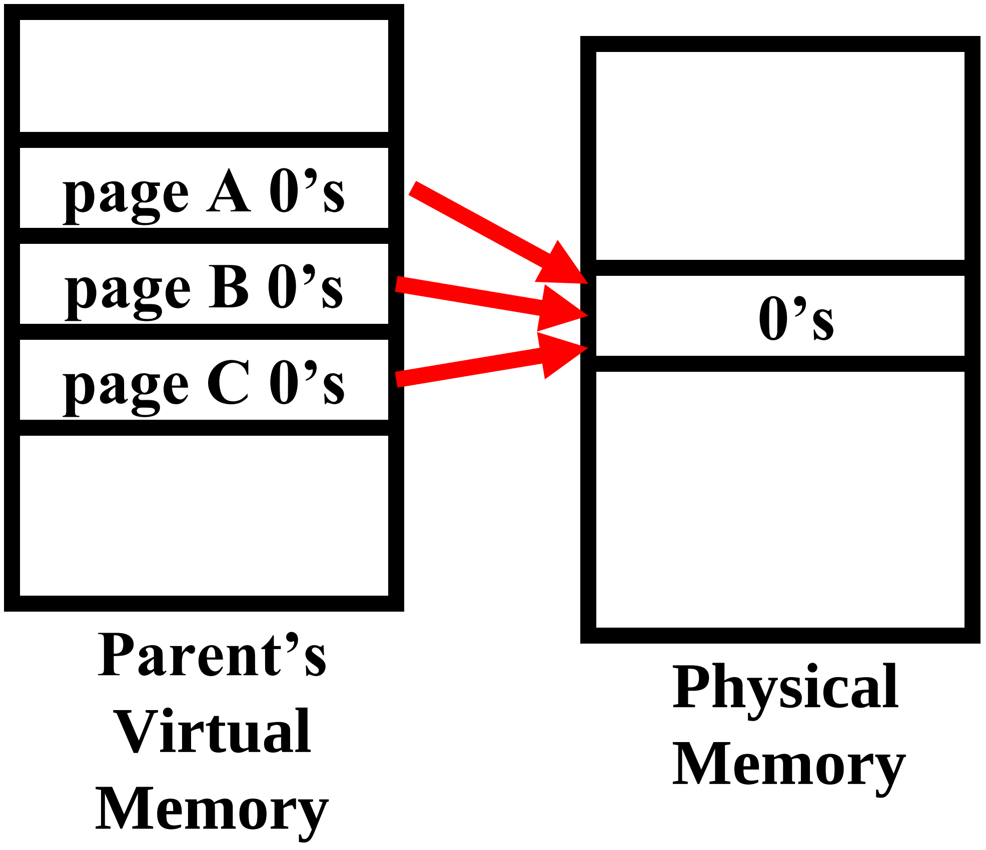
- It is used to allocate space for bss, stack and sbrk() for the heap.
- When allocating memory using sbrk or mmap with the MMAP_ANON flag, all the VM pages in this mapping will map to a single page in RAM that has zeroes and that is read only.
- When a page is modified the OS creates a copy of the page (copy on write) and retries the modifying instruction
- This allows fast allocation. No RAM is initialized to O's until the page is modified
- This also saves RAM. only modified pages use RAM.

4. Allocate zero-initialized memory.

- This is implemented by making the entries in the same page table point to a page with 0s and making the pages read only.
- An instruction that tries to modify the page will get a page fault.
- The page fault allocates another physical page with 0's and updates the page table to point to it.
- The instruction is retried and the program continues as it never happened.

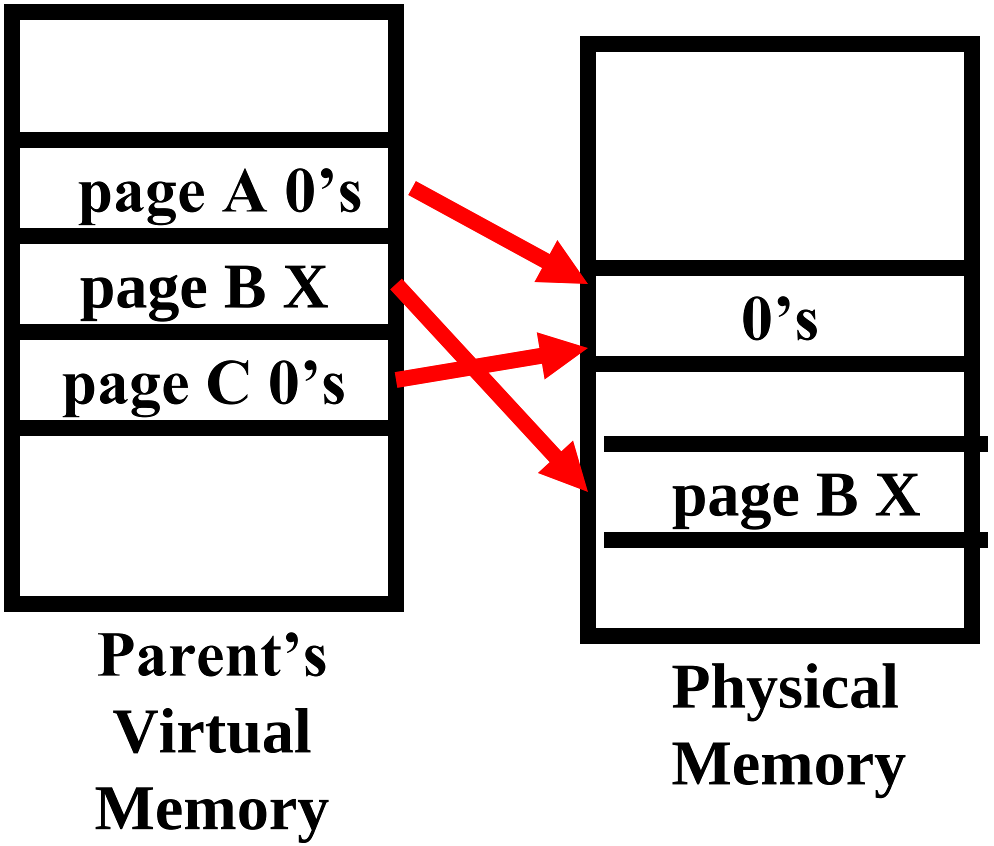
4. Allocate zero-initialized memory.

- After allocating zero initialized memory with sbrk or mmap, all pages point to a single page with zeroes



4. Allocate zero-initialized memory.

- When a page is modified, the page creates a copy of the page and the modification is done in the copy.

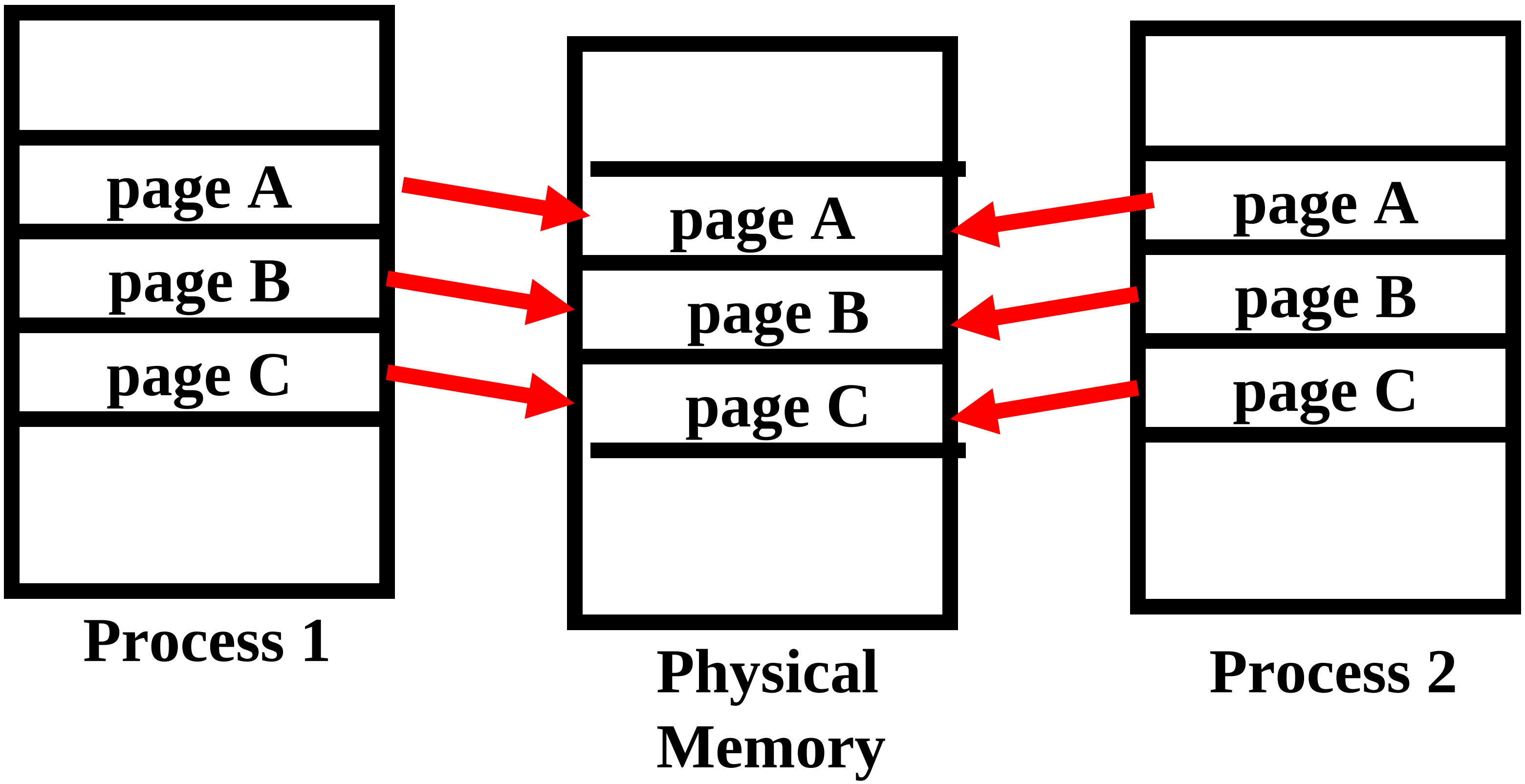


5. Shared Memory

- Processes may communicate using shared memory
- Both processes share the same physical pages
- A modification in one page by one process will be reflected by a change in the same page in the other process.

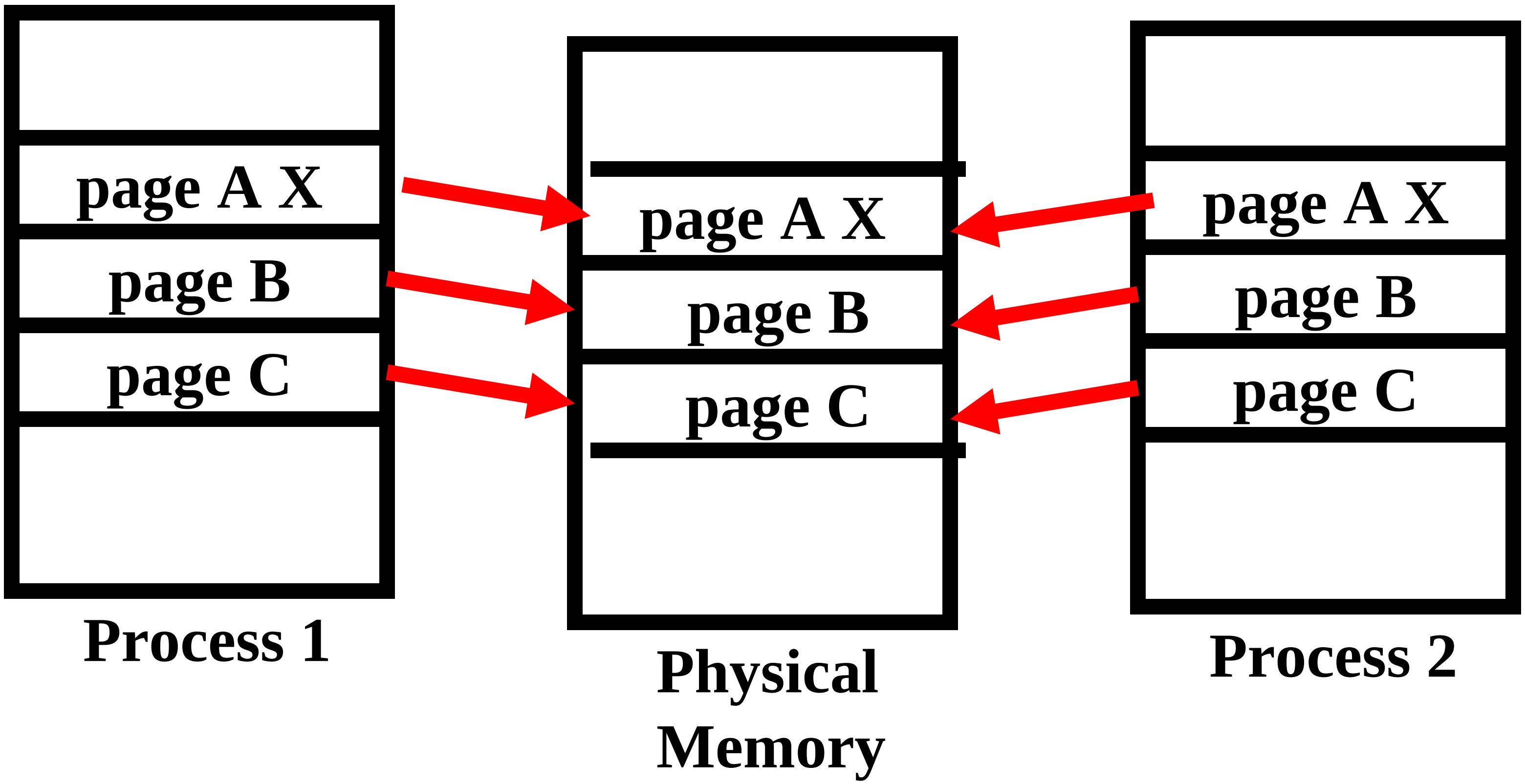
5. Shared Memory

Processes that communicate using shared memory will share the same physical pages.



5. Shared Memory

When a page is modifed, the change will be reflected in the other process.



Caches and Caching

Memory Cache and Caching

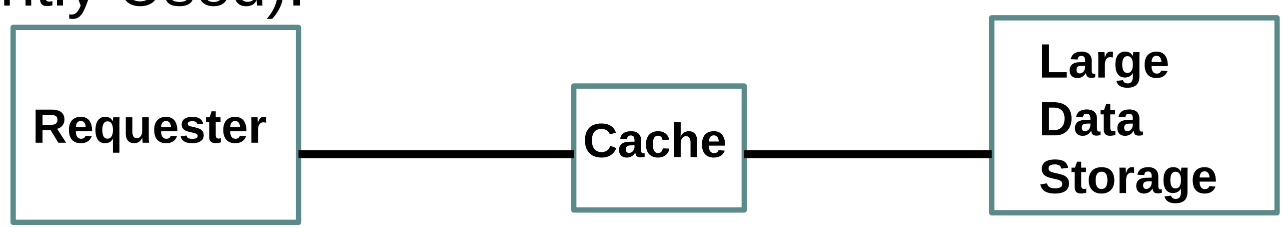
- Caching is associated to two concepts:
  - **Caching** – It is an **optimization technique** that speeds up access to slow storage by storing frequently accessed items in fast storage.
  - **Memory Cache** – It is a type of **fast memory storage** between the CPU and RAM that stores frequently accessed memory items and speeds up memory access.

Caching as an Optimization Technique

- Caching as an optimization technique is used in many instances:
  - **Physical Memory Cache** – Speeds up access to RAM
  - **Web Cache** – Speeds up access to web pages by storing a copy of the frequently accessed pages locally in the computer.
  - **Disk Buffers** – Store in RAM the disk blocks that are frequently accessed in the computer.
  - **VM** – Virtual Memory. Frequently accessed pages are in RAM and rest in Disk.
  - **TLB** – Translation Lookaside Buffer. Frequent VM to Physical address translations are stored in the TLB

Caching as an Optimization

- Cache Hit
  - Request is satisfied from cache,
  - No need to access Large Data Storage
- Cache Miss
  - Request cannot be satisfied from cache,
  - Item is retrieved from Large Data Storage,
  - A copy is placed in cache.
  - If cache is full, an item not used recently in the cache is removed to make room for the new item (LRU-Least Recently Used).



Cache Hit and Miss Ratio

- Hit Ratio (r)
  - Percentage of requests satisfied from cache., between 0 and 1.
- Miss ratio (1-r)
  - Percentage of requests not satisfied from cache.
  - Time = r * Thit + (1-r)*Tmiss   Thit < Tmiss
- As the Hit Ratio (r) increase, the access time decreases.
- In the worst case (r=0) the cost will be the same as not having cache at all.
- In the best case (r=1), all accesses will be in the cache.

Locality of Reference

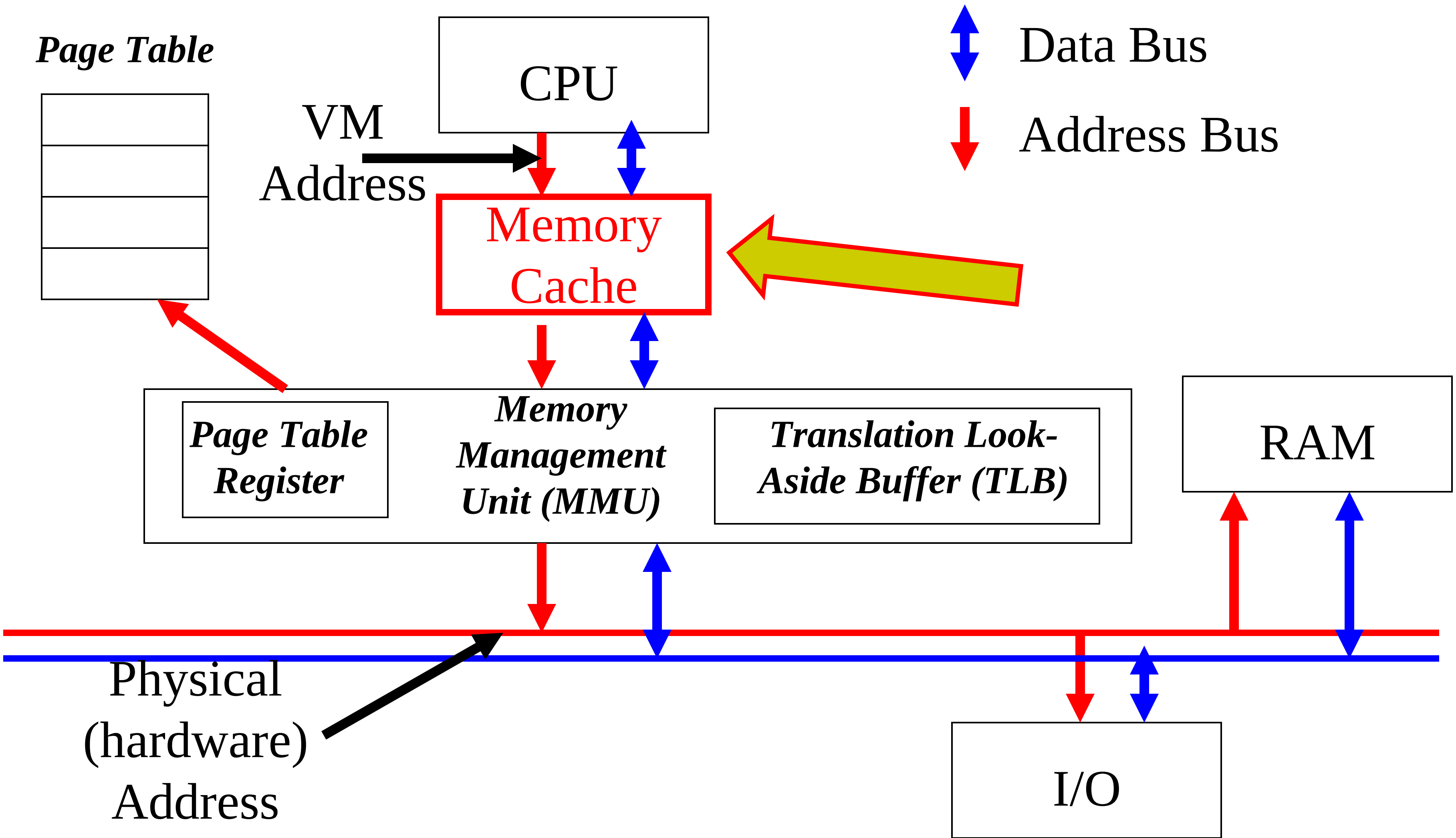
- Locality of Reference refers to repetitions of the same request.
- A **High Locality of Reference** means many repetitions.
- A **Low Locality of Reference** means few repetitions.
- In order for cache to work, we need to have a High Locality of Reference.

Preloading (Prefetch) Caches

- It is an optimization technique where items are stored in the cache **before** the request arrives.
- Works when items are accessed in related groups.
  - When loading a web page, load also the images in the web page.
  - When accessing an item in a struct, also fetch the other items in the struct.
  - When reading a program instruction, also read the instructions that follow.

Memory Cache

- Located between the CPU and RAM



Types of Memory Cache

- It depends how the handle Write Operation
- Write-Through
  - Place a copy of the item in cache
  - Write a copy to Physical Memory
- Write-Back
  - Place a copy of item in cache
  - Write the copy to RAM only when necessary
  - Since multiple processors may have different caches, a **Cache Coherence** mechanism is needed to make sure the cache is consistently updated.

Multi-Level Cache

- L1 Cache
  - Built inside the CPU
- L2 Cache
  - External to the CPU
- L3 Cache
  - Built into RAM

Instruction and Data Caches

- Instructions and Data are both stored in RAM
- Instructions are mostly accessed Sequentially, and they have a High Locality of Reference. For example, there are functions that are executed more than others.
- Data is accessed more randomly but also has some locality of reference. For example, the stack is accessed repeatedly more than some sections in the Heap.
- Some computers separate instruction cache and data cache. Modern computers use the same cache for both instructions and data.

Caching Virtual Memory

- L1 Cache could cache physical Memory or Virtual memory.
- If the VM address is used, then multiple processes may use the same address to store different items.
- If this is the case, the cache also includes the Process ID (PID) as part of the address to disambiguate.

Cache Lines

- The RAM is divided into Cache Lines (also called Blocks).
- The typical size of a Cache Line (Block) is 64 bytes.
- During a Cache Miss, the memory cache fetches an entire Cache Line (64 bytes), even when the CPU only needs 1 byte.
- Prefetching 64 bytes helps sequential access of memory like fetching instructions, and access of data structures that are closed together like structs and local variables.

Cache Technologies

- There are two types of Cache Technologies:
  - Direct Memory Cache
  - Associative Memory Cache
- Direct memory Cache,
  - The memory is divided into blocks (Cache Lines) and each Block has a number.
  - N blocks in memory are grouped into tags.

Direct Cache Tag and Blocks

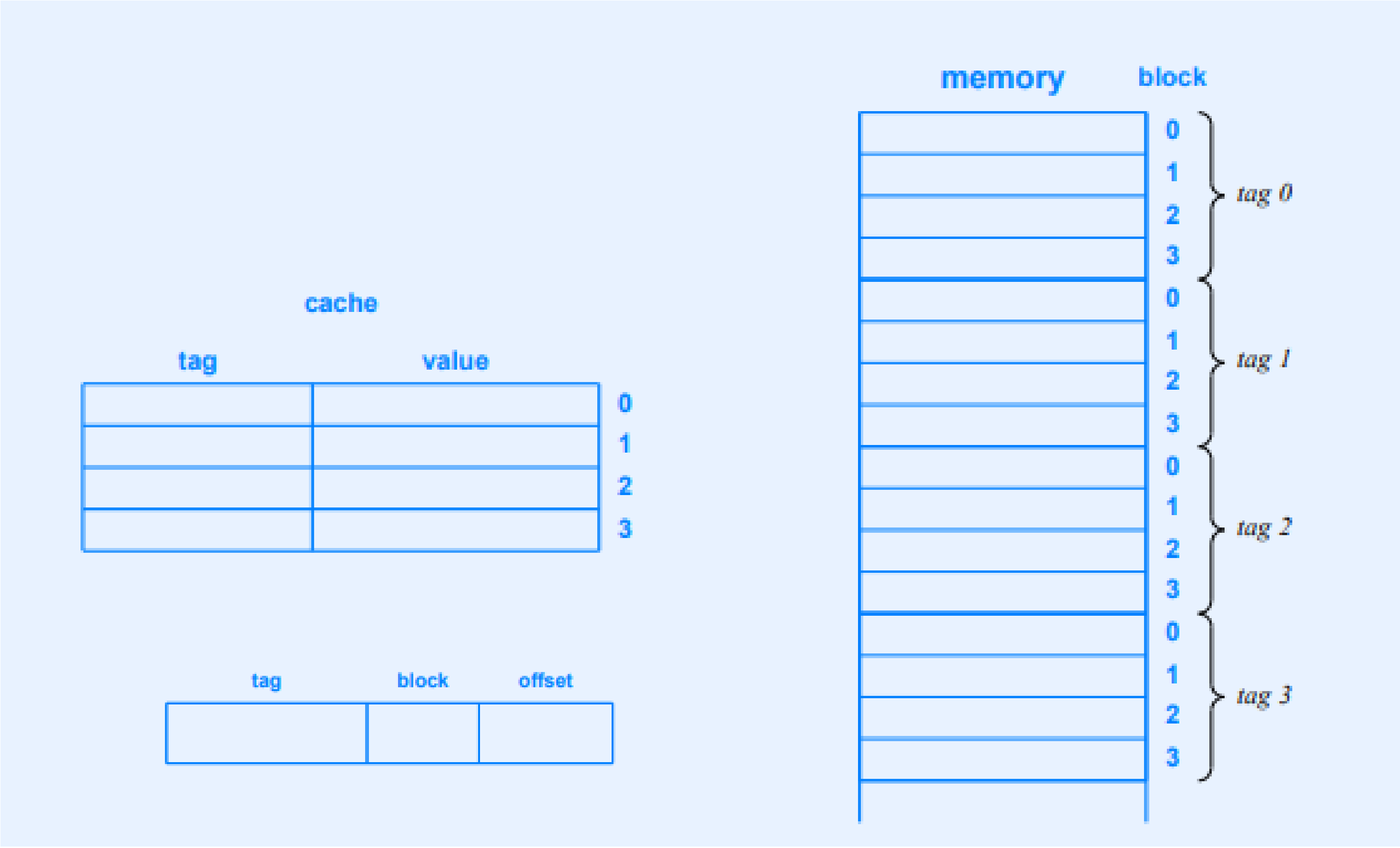


Illustration from Computer Architecture Book by D. Comer

Direct cache Steps

Input: Memory Address

Output: Data in that address

Algorithm:

- Obtain tag t, block number b, and offset
- Examine Tag b in slot b in cache. If tag matches, extract the value from slot b in cache.
- If tag does not match, read the item in memory and copy data in slot b and replace the tag with t.
- Problem: Two items with the same block number cannot be in cache at the same time.

Set associative Cache

- Generalization of Direct Memory Cache.
- Uses parallel hardware.
- Maintains independent caches:

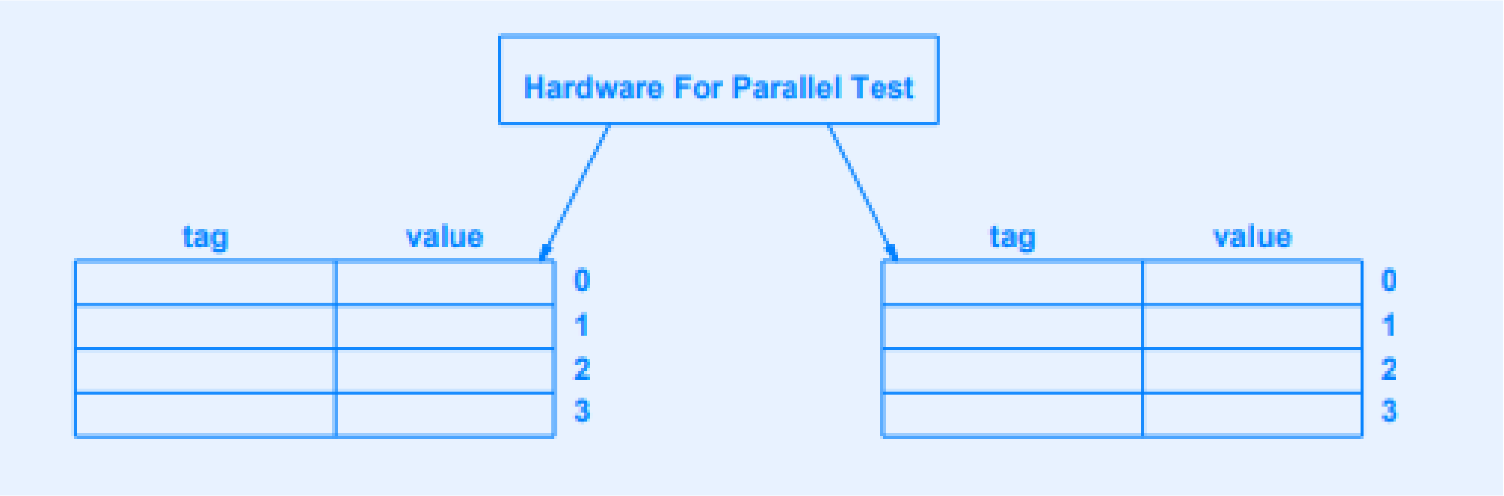
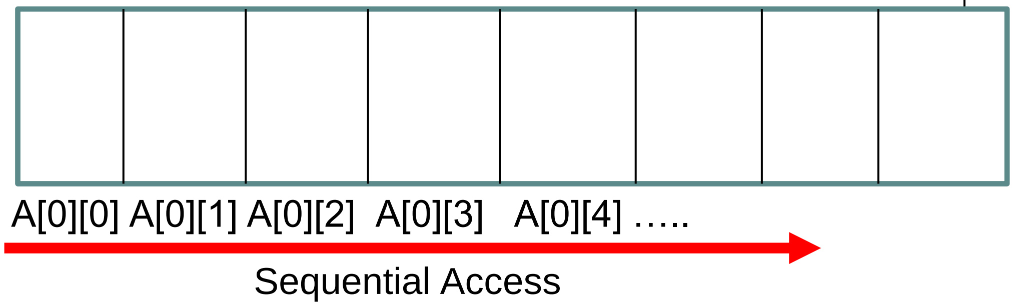


Illustration from Computer Architecture Book by D. Comer

Memory Cache Effect in Performance

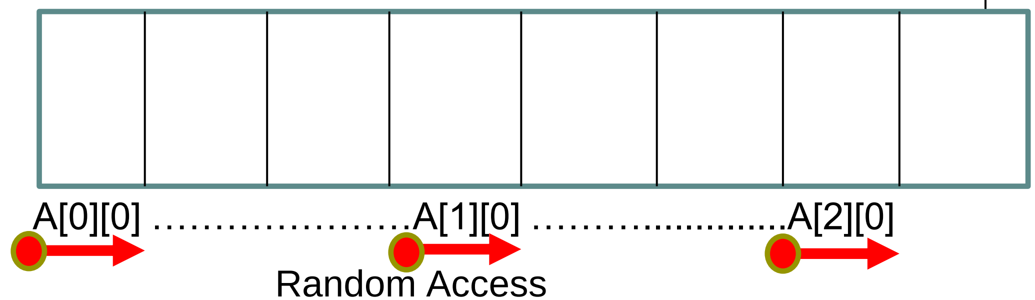
```
// cache1.c
#define M 10000
#define N 10000
long array[M][N];
int
main()
{
        for (int i = 0; i < M; i++) {
                for (int j = 0; j < N; j++) {
                        array[i][j] = j;
                }
        }
}
```




```
grr@data:~/cs250/Summer2020$ gcc -o cache1 cache1.c
grr@data:~/cs250/Summer2020$ time ./cache1
real    0m0.584s
user    0m0.260s
sys     0m0.324s
```

Program is faster because cache access is sequential and there will be fewer cache misses.

Memory Cache Effect in Performance




```
// cache2.c
#define M 10000
#define N 10000
long array[10000][10000];
int
main()
{
        for (int j = 0; j < N; j++) {
                for (int i = 0; i < M; i++) {
                        array[i][j] = j;
                }
        }
}
grr@data:~/cs250/Summer2020$ gcc -o cache2 cache2.c
grr@data:~/cs250/Summer2020$ time ./cache2
real    0m1.396s
user    0m1.046s
sys     0m0.349s
```

Program is slower because cache access is random and therefore there will be more cache misses.

System Calls and Buffering

```
#include <stdio.h>
int
main()
{
	for (char c = 'A'; c < 'Z'; c++) {
		putc(c, stdout);
	}
	putc('\n', stdout);
}
grr@data:~/cs250/Summer2021$ strace ./buffer
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 8), ...}) = 0
brk(NULL)                               = 0x562455464000
brk(0x562455485000)                     = 0x562455485000
write(1, "ABCDEFGHIJKLMNOPQRSTUVWXY\n", 26ABCDEFGHIJKLMNOPQRSTUVWXY
) = 26
exit_group(0)                           = ?
```

Cache and Caching

- Continue Book Class slides
- http://www.cs.purdue.edu/homes/cs250/LectureNotes/book-slides.pdf
- Slide 502
- Chapters XII, XIII, XIV, XV, XVI, XVII (12 , 13, 14, 15, 16, 17).

Statistical Profiling

- At regular intervals it samples what the program is doing.
- It usually happens during context switch with the timer interrupt so there is little overhead.
- The context switch handler records the Program Counter at the time of the context switch.
- The profiler creates a histogram of the number of samples and the function executed in each sample.
- The function with the most samples is the one that is consuming most of the CPU time.
- Execution Profiler Programs with Statistical Profiling:
  - Prof and GProf (Traditional UNIX Profiler)

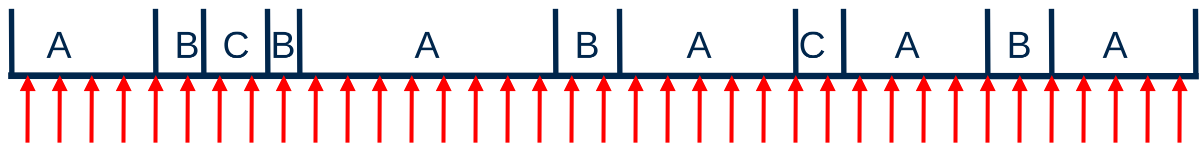
Statistical Profiling

A() {
…
}

B() {
…
}

C() {
…
}

Samples:



T=0.01Secs

A = 25 samples x .01secs/sample = .25secs = 67%

B = 8 samples x .01secs/sample = .08secs = 20%

C = 4 samples x .01secs/sample = .04 secs = 10%

Total=.37

CS250 Computer Architecture Writing the Simple-C Compiler for X86-64

Gustavo Rodriguez-Rivera
Purdue University

Regular Expressions

- Regular expressions describe a family of strings.
- `“abc”` :
  - Represents string abc
- `ba*` :
  - all strings that start with b followed by zero or more a’s .
  - a* means zero or more a’s.
  - Example: b, ba, baaa,
- `[A-Z]*`
  - all strings formed by zero or more upper case letters.
  - [A-Z] means all uppercase letters
  - Example: ZXY, XYY, ABC, AAABBHHH, ZZZ, “” (empty string)
- `-?[0-9][0-9]*`
  - All integers negative or positive. Matches at least one digit.
  - ? Means optional (one or empty string). Sign is optional.
  - Example: 123, -23, -9.

Regular Expressions

- `[^ \n\t]*`
  - All characters except space, new line or tabs including empty string
  - ^ at the beginning means everything that is not.
- [A-Za-z][A-Za-z0-9]*
  - A string that starts with a letter followed by zero or more letters or characters
  - This is the regular expression for an identifier for a variable or function name.
  - Examples: main, i, area4,

Regular expressions are used to represent tokens, that is a sequence of characters.

Context Free Grammars

- Context Free grammars are used to define the structure of a program.
- Regular expressions are not powerful enough to represent the grammar of a program, that is why we use Context Free grammars
- A context free grammar also defines a family of strings:
- A context free grammar is defined by a list of grammar rules.
- The grammar rules may contain Terminals (Tokens) or Non-terminals, that are the names of other grammar rules.
- We use lower case for non-terminals (like x) and uppercase for Terminals (like A,B)
- Example:
  - goal->x
  - x->A
  - x->x B

This may produce strings: A, AB, ABB, ABBB, ABBBB

But it cannot produce strings such as AA, XYZ, AAA, etc.

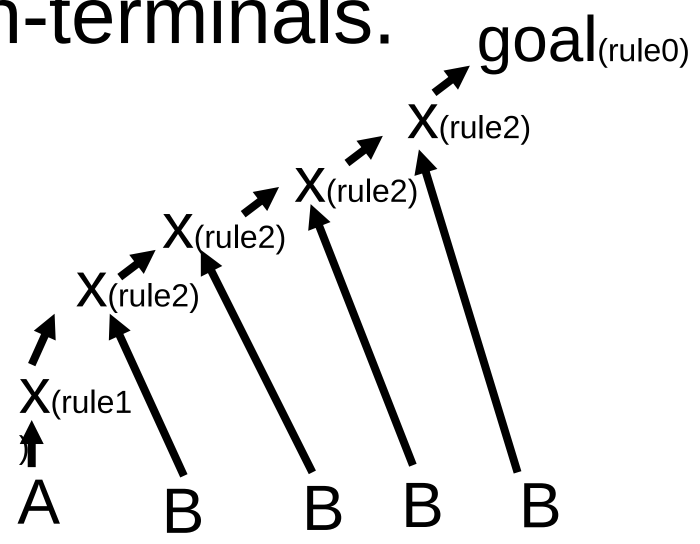
Context Free Grammar lists

- Sequence of terminals:
  - goal->x
  - x->ABC

  Strings: ABC
- One or more elements
  - goal->x
  - x-> A
  - x->xA
  - Strings: A, AA, AAA, AAAA …
- Zero or more elements: epsilon represents the empty string.
  - goal->x
  - x-> ***epsilon***
  - x->xA
  - Strings: “”, A, AA, AAA, AAAA …

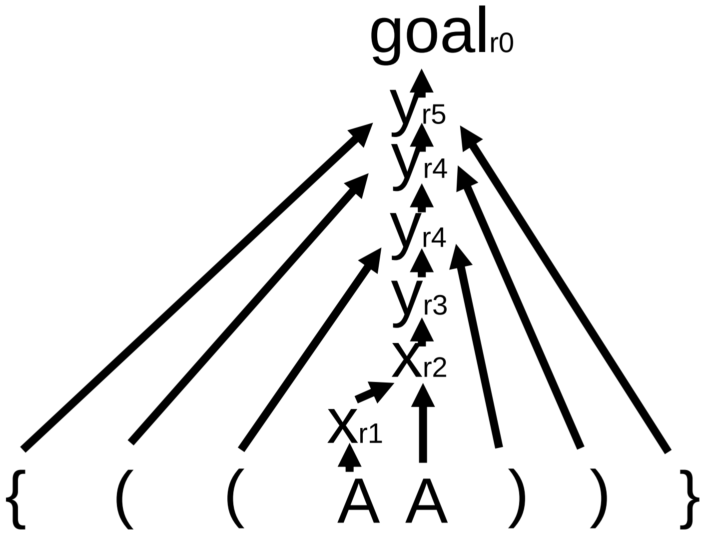
Context Free Grammars

- How do we prove that ABBBB matches the grammar?
- We can build a “parse tree”. At the bottom we place the terminals and in the internal nodes we place the non-terminals.
  - goal->x (rule 0)
  - x->A (rule 1)
  - x->xB (rule 2)



Context Free Grammars

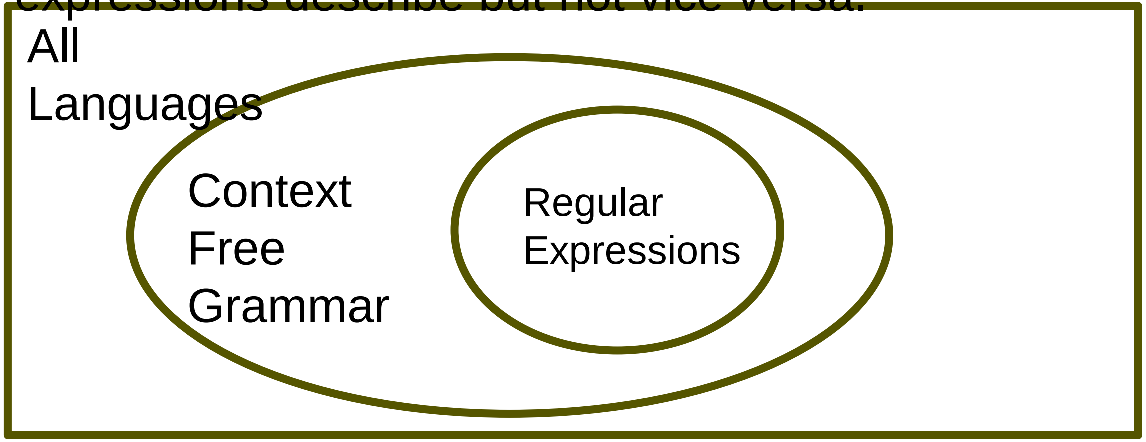
- Context free grammars are more powerful than regular expressions
- We can write a grammar of strings with matching parenthesis and curly brackets
  - goal->y (rule 0)
  - x->A (rule 1)
  - x->xA (rule 2)
  - y->x (rule 3)
  - y->(y) (rule 4)
  - y->{y} (rule 5)



- Parse tree for "{((AA))}" matches the grammar.

Context Free grammars

- Context Free Grammars can describe all languages that regular expressions describe but not vice versa.



Language == Family of strings

- We will use Regular expressions to describe tokens, in simple.l
- We will use Context Free Grammars to describe the SimpleC language in simple.y

Using Context Free Grammars for Computer Languages

We can describe a computer language using Context Free Grammars.
For example, a program is a sequence of functions and global variables:

goal -> program

program-> function_or_var_list

function_or_var_list-> function_or_var_list   function

function_or_var_list-> function_or_var_list   global_var

function_or_var_list -> ***epsilon***

function -> var_type WORD LPARENT arguments RPARENT compound_statement

Using Context Free Grammars for Computer Languages

In Yacc, we can group the rules that define the same non-terminal. The previous can be written for Yacc as follows:

```
goal :
        program
        ;
program:
        function_or_var_list
        ;
function_or_var_list:
          function_or_var_list   function
        | function_or_var_list   global_var
        | /* epsilon */
        ;
function :
        var_type WORD LPARENT arguments RPARENT compound_statement
        ;
```

Lab: Writing a Simple Compiler

- In this lab you will write a compiler for "Simple C"
- This language is a reduced version of "C".
- We will concentrate on generating the assembly language code.
- We will cover superficially the theory of parsing and the use of Lex and Yacc

Simple C

- Subset of C
- Only the following types are supported:

  long

  long*

  char

  char*

  void
- Also it supports constructions such as if/else, while, do/while, for.
- The program consists of a declaration of functions and variables like in "C".
- Also you can call "C" functions from Simple C as long as the arguments they use are char* and long (or int).

Example Simple “C” program

```
long fact(long n) {
    if (n==0) return 1;
    return n*fact(n-1);
}

void main()
{
    printf("Factorial of 5 = %ld\n" , fact(5));
}
```

Building a Compiler

- To help in the development of compilers, tools such as Lex and Yacc have been created.
- With these tools, the programmer concentrates only in the grammar and the code generation.

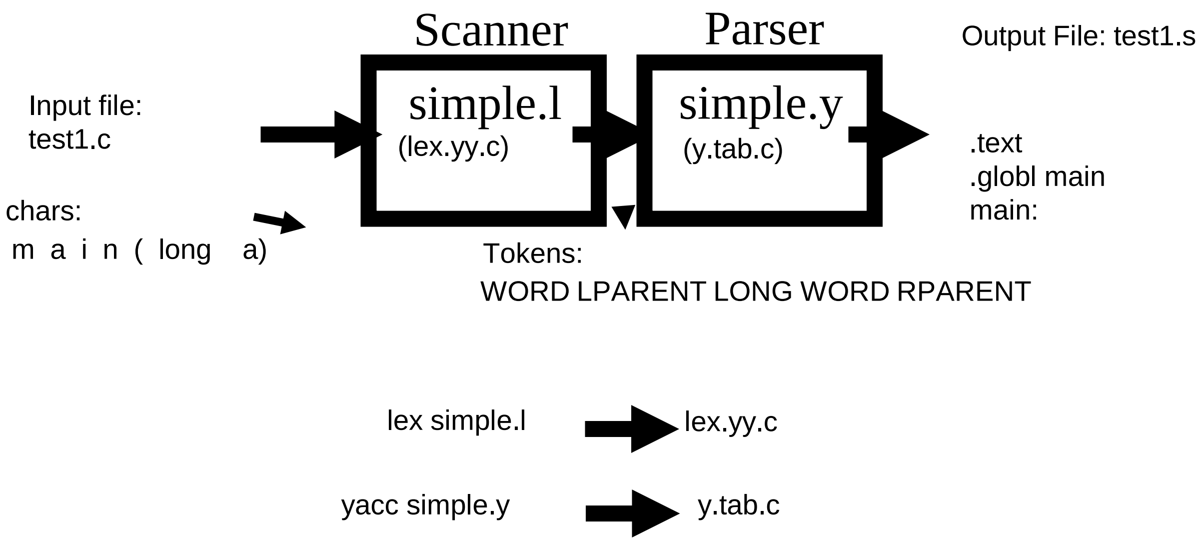
Lex

- Lex
  - takes as input a file *simple.l* with the regular expressions that describe the different tokens.
  - It generates a scanner file “lex.yy.c” that reads characters and forms tokens or words that the parser uses.

Yacc

- Yacc
  - Takes as input a file simple.y with the grammar that describes the language.
  - This file also contains “actions” that is “C” code that describes how the code will be generated while parsing the code.
  - It generates a parser file called “y.tab.c” that reads the tokens and parses the program according to the syntax.
  - When it reaches an action in the syntax tree, it executes that action

Lex and Yacc Interaction



Lex Input file simple.l

- It contains the regular expressions that describe the different tokens

```
"return" {
        return RETURN;
    }

[A-Za-z][A-Za-z0-9]*  {
        /* Assume that file names have only alpha chars */
        yylval.string_val = strdup(yytext);
        return WORD;
    }
```

Yacc input file simple.y

- It contains the grammar that describes the language.
- It also contains actions or c code that will be executed after parsing specific grammar constructions.
- It also includes the main() entry point of the compiler.

Yacc input file simple.y

```
program :
        function_or_var_list;

function_or_var_list:
        function_or_var_list function
        | function_or_var_list global_var
        | /*empty */
        ;
                                      long i; long k; void main(){ …} void foo(){…}
function:
         var_type WORD
         {
                 fprintf(fasm, "\t.text\n");
                 fprintf(fasm, ".globl %s\n", $2);
                 fprintf(fasm, "%s:\n", $2);
         }
         LPARENT arguments RPARENT compound_statement
         {
                 fprintf(fasm, "\tret\n");
         }
        ;
```

Code generation

- You will need to add more actions to generate the code.
- An action is a portion of code such as

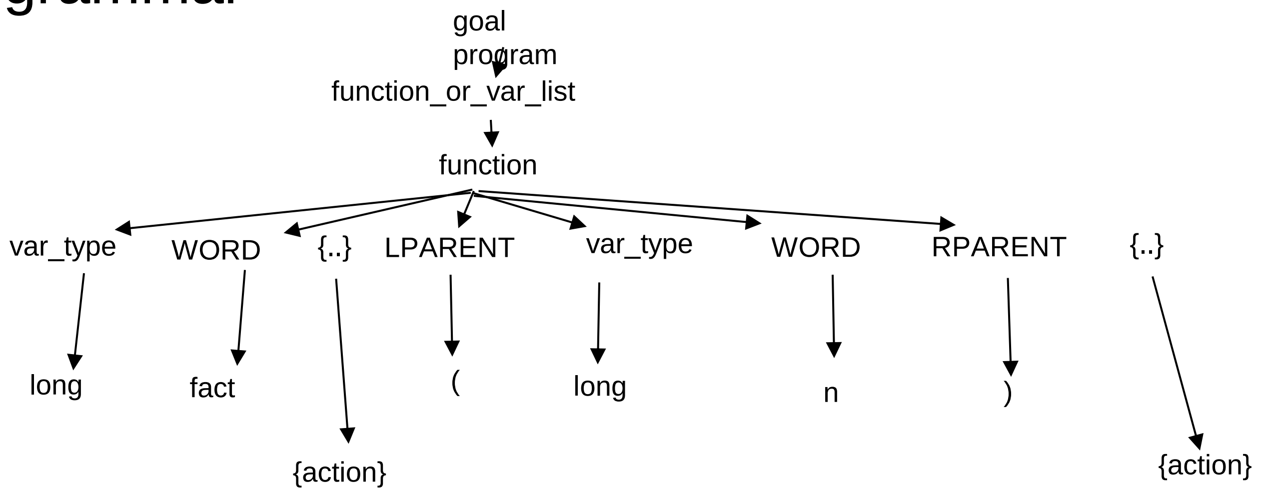
```
{
        fprintf(fasm, "\tret\n", $2);
}
```

that is embedded in the grammar.

- This portion of code is executed when the parser reaches that point.

Parsing tree

- The parser tries to parse the input according to the grammar



Parsing Expressions

- To describe expressions in SimpleC, We need the hierarchy of logical, equality, relational, additive, multiplicative expressions to take into account the operator precedence.

```
expression :
        logical_or_expr
        ;
logical_or_expr:
        logical_and_expr
        | logical_or_expr OROR logical_and_expr
        ;

logical_and_expr:
        equality_expr
        | logical_and_expr ANDAND equality_expr
        ;
```

Parsing Expressions

```
equality_expr:
        relational_expr
                | equality_expr EQUALEQUAL relational_expr
                | equality_expr NOTEQUAL relational_expr
                ;
relational_expr:
        additive_expr
                | relational_expr LESS additive_expr
                | relational_expr GREAT additive_expr
                | relational_expr LESSEQUAL additive_expr
                | relational_expr GREATEQUAL additive_expr
                ;
```

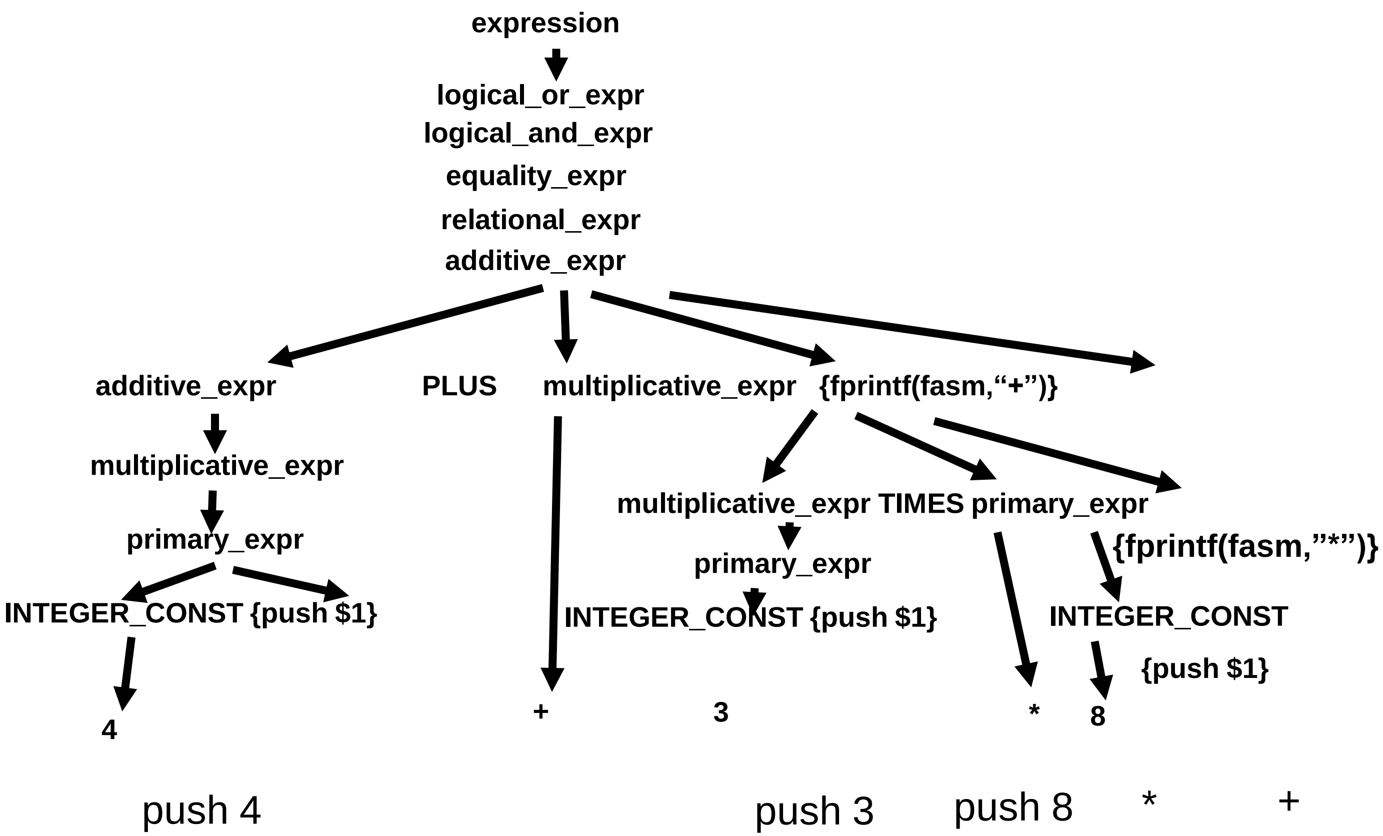
Parsing Expressions

```
additive_expr:
        multiplicative_expr
                | additive_expr PLUS multiplicative_expr {
                        fprintf(fasm, "\t# +\n");
                }
                | additive_expr MINUS multiplicative_expr
                ;

multiplicative_expr:
        primary_expr
                | multiplicative_expr TIMES primary_expr {
                        fprintf(fasm, "\t# *\n");
                }
                | multiplicative_expr DIVIDE primary_expr
                | multiplicative_expr PERCENT primary_expr
                ;
```

Parsing Expressions

```
primary_expr:
        STRING_CONST {
                // Add string to string table.
                // String table will be produced later
                string_table[nstrings]=$<string_val>1;
                fprintf(fasm, "\tmov $string%d, %%rdi\n",
nstrings);
                nstrings++;
        }
        | call
        | WORD
        | WORD LBRACE expression RBRACE
        | AMPERSAND WORD
        | INTEGER_CONST {
                fprintf(fasm, "\t# push %s\n", $<string_val>1);
        }
        | LPARENT expression RPARENT
        ;
```

How expressions are parsed



Generating Code for Expressions

- Since the compiler will only parse the sources once, the easiest code to generate is the code for a stack-based machine.
- However, a stack-based machine is slow.
- We will optimize this by using registers for the bottom entries of the stack.

Example of stack-based machine

- Arithmetic expression:

  4+3*8

- Equivalent in stack-based machine:

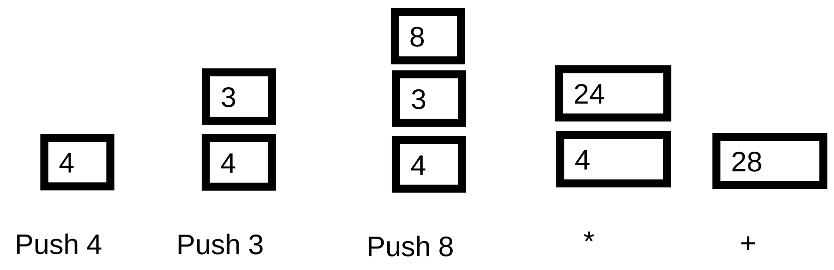
  push 4

  push 3

  push 8

  *

  +



- This form of evaluation is also called RPN (Reverse Polish Notation)
- The compiler can generate this stack-based code using a post-order traversal of the parse-tree.

Expressions Code Generation

- You will use a Stack Virtual Machine.
- The bottom elements in the stack will be stored in registers to speed up access.
- You will need to save these registers at the beginning of the function and restore them before returning.

Stack Representation

Stack Position	Register/Memory
0	rbx
1	r10
2	r13
3	r14
4	r15
>=5	Use the execution stack

Stack Operations

- Depending on the stack position, the push or pop instruction will use a different register.
- Example: 4+3*8

```
movq $4,%rbx        # push 4. Use %rbx
movq $3,%r10        # push 3. Use %r10
movq $8,%r13        # push 8. Use %r13
imulq  %r13,%r10    # * = Multiply 2 top values.
                    #   Push result.
addq %r10,%rbx      # + = Add 2 top values.
                    #   Push result
movq $rbx, $rax     # move result to %rax for use in
                    # statements
```

- **See starting code in `simple.y`**

Implementing Variables

- Your compiler will handle three type of variables:
  - Global variables
  - Local Variables
  - Arguments

Implementing Declaration of Global Variables

- The declaration of global variables are parsed in the rule:

```
global_var:
        var_type global_var_list SEMICOLON;

global_var_list: WORD
        | global_var_list COMA WORD
        ;
```

- Insert the actions {…} to
  - reserve space
  - add the variable to the global variable table.

Creating Space for Global Variables

- Global variables are stored in the data section.
- Generate code that way:

Example:

Simple C:

```
long g;
```

Assembly:

```
.data
 .comm g, 8
```

Getting a Value from a Global Variables

- The parse rule that should generate the code for getting the value of a global variable is:

```
primary_expr:
   ….
   WORD {
      char * id = $<string_val>1;
     lookup id in local variables table
      if id is a local var {
          read local var from runtime stack and push into register stack.
               (We will see this later).
      }
      else {
        lookup id in global var table
        if id is a global var {
          Generate code to read global var and push it to stack
          fprintf(fasm, "movq %s, %%%s\n", id, regStk[top]);
          top++;
         }
   }
     …
```

Saving into a global variable

- Storing into a global variable is implemented in the following rule

```
assignment:
      WORD EQUAL expression {
                 // Code for a assignment
         char * id = $<string_val>1;
         if (id is local var) {
            // we will see later
         }
         else if (id is a global var) {
           // Generate code to save top of the stack
           // in global var
          fprintf(fasm, "movq %%rbx,%s\n", id);
          top = 0;
        }
      }
```

Getting a Value from a Global Variables

- Example:

  Simple C:

```
x = 5 + g;
```

  Assembly

```
movq $5, %rbx      # push 5
movq g,%r10        # push g (printed by code
                        #          shown before)
addq %r10,%rbx     # add and push result
                   #    to top of stack
movq %rbx, x       # Save result into x
```

Implementing Declaration of Local Variables

- Declaration of local variables should be done in the production

```
local_var:
        var_type local_var_list SEMICOLON;

local_var_list:
          WORD
        | local_var_list COMA WORD
        ;
```

Implementing Declaration of Local Variables

- Local variables are stored in the stack.
- We need to reserve stack space at the beginning of the function using
  - **subq $<space>, %rsp**
  - **Where <space> is the space reserved that needs to be restored before leaving the function.**
- We do not know how much space to reserve.

Implementing Declaration of Local Variables

- Three approaches:
  - Reserve a constant maximum stack space all the time Example: 256 bytes, enough for 32 variables.
  - Use a “.include testx.h” at the top of the assembly file, write that constant in testx.h as “.set _funx_space, 24” (assuming 24 bytes of space for funx) when we know space to reserve. Use “subq $ _funx_space, %rsp” to reserve the space.
  - You may not need to use an include. Insert “**subq $** _funx_space**, %rsp” and then at the end of the function, once you know the space,** call “.set _funx_space, 24” at the end of the function.
- All these approaches are OK for the purpose of this project.

Implementing Declaration of Local Variables

- Remember that the argument registers are overwritten during a function call.
- You need to save the argument registers in the stack at the beginning of the function.
- Hint:
  - Add the arguments to the local variable table at the beginning of the function and treat the arguments as local variables.

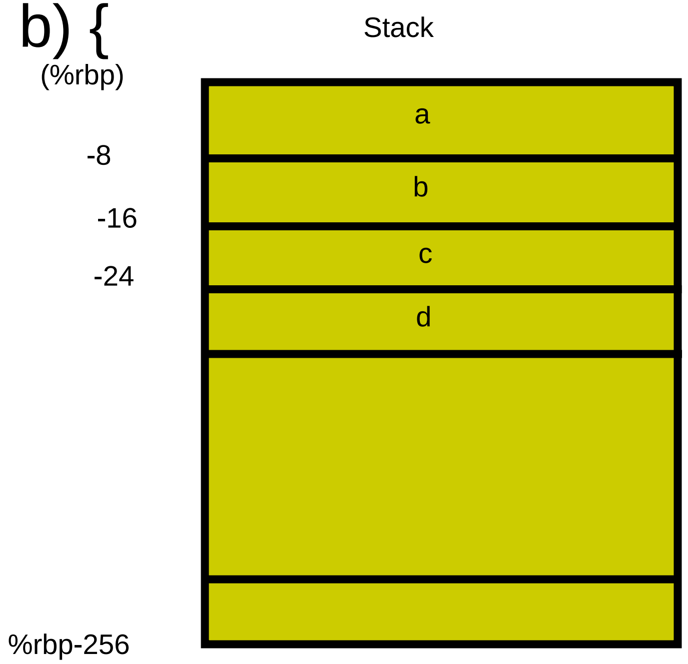
Implementing Declaration of Local Variables

Example:

```
long add(long a, long b) {
    long c,d;
    c = 5;
    d = c + a*b;
    return d;
}
```

To push b to top of register stack:

movq -16(%rbp),%rbx



Implementing Declaration of Local Variables

```
local_var_list: WORD {
  // first local variable
  assert(nlocals < MAXLOCALS);
  local_vars_table[nlocals]=$<string_val>1;
  nlocals++;
}
| local_var_list COMA WORD {
  assert(nlocals < MAXLOCALS);
  local_vars_table[nlocals]=$<string_val>3;
  nlocals++;
}
```

Generating code for while()

```
long i = 0;
main()
{
        while (i<5) {
                i= i + 1;
                printf("%d\n", i);
        }
}
```

Generating code for while()

```
            .data
        .comm i, 8

            .text
            .globl main
main:…
                                    #while (i<5) {
while_1:                                    #    expression i<5
            movq i, %rbx            # push i
            movq $5, %r10                   # push 5

            movq $0,%rax                    # Zero %rax
            cmpq %r10,%rbx                  # compare top of the stack (rbx-r10)
            setl %al                    # Set byte if less
             # See http://www.amd.com/us-en/assets/content_type/
white_papers_and_tech_docs/24592.pdf page 55 or
https://www.felixcloutier.com/x86/setcc.html
            movq %rax,%rbx                  # Put result back to top

            cmpq $0, %rbx                   # Compare top of the stack with 0
            je after_while_1                # Jump after while if not true
```

Generating code for while()

```
                    # Body of while
        movq i,%rbx          #   i = i+1
        movq $1,%r10
        addq %r10,%rbx
        movq %rbx,i
                             #   printf("%d\n,i) ;
        movq $str1, %rbx          # Arg1 of printf
        movq %rbx, %rdi

        movq i,%rbx  # Arg2 of printf
        movq %rbx, %rsi

        movq $0,%rax              # Extra 0s for printf
        call printf               # Call printf

        jmp while_1     # } // while
after_while_1:
        ret
        .text
str1:
        .string "i=%d\n"
```

Parsing Arguments for Calls

- When parsing argument to calls let the parser push the expressions to the register stack.
- Do not initialize top at every argument.
- The arguments will be saved in the register stack until they are copied to the register arguments.

Parsing Arguments for Calls

```
Simple C:
  printf("compute(3,4)=%d\n", compute(3,4));
Assembly:
  movq string0, %rbx # push string0  - printf's arg1
  movq $3, $r10      # push 3        - compute's arg1
  movq $4, $r13      # push 4        - compute's arg2

                     # Copy from stack to arg regs top==3
  movq $r13, $rsi    # pop into register for arg2  top==2
  movq $r10, $rdi    # pop into register for arg1  top==1
  call compute
  movq %rax, %r10    # Push return val to stack    top==2

  movq $r10, $rsi    # pop into register for arg2  top==2
  movq $rbx, $rdi    # pop into register for arg1  top==1

  movl $0, %eax      # Call printf
  call printf
```

Parsing Arguments for Calls

- The problem with nested calls is that a single "nargs" variable is not enough to keep count of the number of arguments.
- The solution is to store an "nargs" into the call_arg_list nonterminal to make the nargs local to the function parsed.
- In %union add:

```
%union          {
        char   *string_val;
      int    nargs;
  }
```

- This will allow adding a new type

Parsing Arguments for Calls

- Modify call_arg_list to count the arguments. The $<nargs>$ stores a variable nargs local to this rule inside the non-terminal expression that can be used later.

```
call_arg_list:
        expression {
            $<nargs>$ = 1; // Initialize args to 1
}
| call_arg_list COMA expression {
            $<nargs>$++;
 };

call_arguments:  /* Pass up number of args */
          call_arg_list { $<nargs>$=$<nargs>1;}
          | /*empty*/ { $<nargs>$=0;}
      ;
```

Parsing Arguments for Calls

```
call :
        WORD LPARENT  call_arguments RPARENT {
                   int i;
                   char * funcName = $<string_val>1;
                   if (!strcmp(funcName, "printf")) {
                            // printf has a variable number
of args
                            fprintf(fasm, "\tmovl    $0,
%%eax\n");
                   }
                  // Move from top of stack to argument
registers
                fprintf(fasm, "    #Push arguments to
stack\n");
                  for (i=$<nargs>3-1; i>=0; i--) {
                           top--;
                           fprintf(fasm, "\tmovq %%%s,
%%%s\n",
                                     regStk[top],
                                     regArgs[i]);
                  }
                  fprintf(fasm, "\tcall %s\n", funcName);
        }
  ;
```

Final Exam Review

Final Exam Review

- VIII. Assembly Language and Programming
  - X86-Assembly Language
    - Register Assignment
    - Addressing Modes
    - Using the stack
    - Calling Conventions
    - Flow Control
- IX. Memory and Storage
  - Volatile, Non-volatile,
  - Random Access and Sequential Access
  - ROM, PROM, EEPROM
  - Memory Hierarchy
- XI. Virtual Memory
  - MMU,
  - Physical and VM Address Memory
  - Address Translation
  - Two-level page table
  - Page Bits
  - Page faults
  - TLB’s
  - Row major and column major computations

Final Exam Review

- XII Caches and Caching
  - Importance of Caching
  - Cache hit and cache miss
  - Locality of reference
  - Worst /Best/Average case cache performance
  - Hit /Miss ratio
  - Multiple levels of cache
  - Preloading caches
  - Write-through and write back cache
  - L1, L2, L3 cache
  - Direct mapping and set associative cache

Final Exam Review

- XIII Input/Output Concepts and Terminology
  - Parallel Interface / Serial Interface
  - Data Multiplexing
- XIV Buses and Bus Architecture
- XV Programmed and Interrupt-Driven I/O
  - Polling ad Interrupts
  - Handling an Interrupt
  - Interrupt Vector
  - Multple levels of interrupts
  - DMA
  - Buffer chaining and Scatter Read and Gather Write

Final Exam Review

- XVI. A Programmers View of I/O and Buffering
  - Upper Half and Lower Half of a Device Driver
  - Character oriented and block oriented devices
  - Buffered input and output.

Final Material to Study

- New Slides
- Old slides
- Everything up to and including chapter XIX in the book.
- Projects
- X86-64
- Assembly Programming materials
- I will ask code fragments of the compiler project.

Extra Slides

Project4
Changing numBuckets

- Using a power 2 will make the % operation in hashing faster.
- The mod operation can be substituted by &

- Example:
  - N=2^4=0x10   x%N= x & 0xF
    50 % 16 = 110010b & 1111b = 0010b = 2
    50%16=2d

- For N=2^20=0x100000
  - x % (2^20)= x & 0xFFFFF

ARM Assembly Language

ARM Architecture

- ARM- Acorn RISC Machine
- ARM is an architecture created by “ARM Holdings”
- ARM Holdings does not manufacture the CPU’s, instead it licenses the design to other manufacturers so they create their own version of ARM.
- ARM has become popular because of mobile computing: Smart phones, tablets etc.
- It is energy-efficient, fast, and simple.
- It still lags in speed compared to the fastest Intel x86 CPUs but it is more energy efficient.

ARM CPUs

- Chips using ARM architecture
  - A4, A5, A6, A7
    - Iphone/Ipad by Apple
  - Qualcomm’s Snapdragon
    - Samsung Galaxy, LG, Nokia Lumia, Sony, Kindle
  - NVIDIA Tegra
    - Windows RT Tablet, Motorola Droid, Motorola Atrix
  - Broadcom, BCMXXX CPUs
    - Samsung Galaxy, Raspberry Pi

ARM Assembly Language

- See:

  http://www.cs.purdue.edu/homes/cs250/LectureNotes/arm_inst.pdf

  and

  http://www.cs.purdue.edu/homes/cs250/LectureNotes/arm-ref.pdf

Example Assembly Program

test1.s:

```
.data                        /* Switch to data section
print_string:
     .asciz "Hello world\n"  /*Define string used in printf*/
                             /* void main() { */
.text                        /*Switch to text section*/
      .global main           /*Export symbol "main"
main:                        /*main function starts here */
      push {r4-r9, fp, lr}   /*save registers used in function */
                                 /* printf("Hello world\n"); */
      ldr     r0, =print_string  /*Load argument for printf*/
      bl      printf           /* Call printf */

                               /* } */
      pop {r4-r9, fp, pc}      /*Restore registers and return */
```

Running the Assembler

```
pi@raspberrypi:~/cs250/lab6-src$ gcc -o test1 test1.s

pi@raspberrypi:~/cs250/lab6-src$ ./test1
Hello world

pi@raspberrypi:~/cs250/lab6-src$
```

Assembly Code in Hexadecimal

pi@raspberrypi:~/cs250/lab6-src$ gcc -Xassembler -a -o test1 test1.s > out
pi@raspberrypi:~/cs250/lab6-src$ vi out

```
ARM GAS  test3.s                            page 1
   1                    .data
   2                    print_string:
   3 0000 48656C6C           .asciz "Hello world\n"  /* Define string used in printf */
   3      6F20776F
   3      726C640A
   3      00
   4                    .text
   5                          .global main
   6                    main:
   7 0000 F04B2DE9            push {r4-r9, fp, lr}
   8 0004 04009FE5            ldr     r0, =print_string
   9 0008 FEFFFFEB            bl      printf
  10 000c F08BBDE8            pop {r4-r9, fp, pc}
  11 0010 00000000
DEFINED SYMBOLS
             test3.s:2      .data:00000000 print_string
             test3.s:6      .text:00000000 main
             test3.s:7      .text:00000000 $a
             test3.s:11     .text:00000010 $d
UNDEFINED SYMBOLS
Printf

The third column is the code generated in hexadecimal.
```

Calling Conventions

- r0 to r3:
  - They are used to pass arguments to a function. r0 is used to return values. (No need to be restored before return).
- r4 to r11:
  - Used to hold local variables. (Need to be restored before return)
- r13 is the stack pointer.
  - Stores return PC and save registers and local vars.
- r14 is the link register. (The BL instruction, used in a subroutine call, stores the return address in this register).
- r15 is the program counter.

Condition Code Flags

- This flags are stored in the PSR- Processor Status Register
- They are updated by the Arithmetic Operations

N = Negative result from ALU flag.

Z = Zero result from ALU flag.

C = ALU operation Carried out

V = ALU operation oVerflowed

Updating the Condition Code Flags

- CMP reg1, reg2
  - Performs reg1-reg2
  - It updates N, Z, C, V
  - No other registers are modified
- TST reg1, reg2
  - Performs reg1 bit-and reg2
  - It updates N,Z
  - No other registers are modified
- Any instruction may modify the flags if “S” is appended to the instruction:
  - Example MOVS reg1, reg2 will update N, Z if reg2 is zero or negative

ARM Instructions

ARM assembly language reference card
MOVcdS reg, arg copy argument (S = set flags)
MVNcdS reg, arg copy bitwise NOT of argument
ANDcdS reg, reg, arg bitwise AND
ORRcdS reg, reg, arg bitwise OR
EORcdS reg, reg, arg bitwise exclusive-OR
BICcdS reg, rega, argb bitwise rega AND (NOT argb)
ADDcdS reg, reg, arg add
SUBcdS reg, reg, arg subtract
RSBcdS reg, reg, arg subtract reversed arguments
ADCcdS reg, reg, arg add with carry flag
SBCcdS reg, reg, arg subtract with carry flag
RSCcdS reg, reg, arg reverse subtract with carry flag
CMPcd reg, arg update flags based on subtraction
CMNcd reg, arg update flags based on addition

ARM Instructions

TSTcd reg, arg update flags based on bitwise AND
TEQcd reg, arg update flags based on bitwise exclusive-OR
MULcdS regd, rega, regb multiply rega and regb, places lower 32 bits into regd
MLAcdS regd, rega, regb, regc places lower 32 bits of rega · regb + regc into regd
UMULLcdS reg`, regu, rega, regb multiply rega and regb, place 64-bit unsigned result into {regu, reg`}
UMLALcdS reg`, regu, rega, regb place unsigned rega · regb + {regu, reg`} into {regu, reg`}
SMULLcdS reg`, regu, rega, regb multiply rega and regb, place 64-bit signed result into {regu, reg`}
SMLALcdS reg`, regu, rega, regb place signed rega · regb + {regu, reg`} into {regu, reg`}
Bcd imm12 branch to imm12 words away
BLcd imm12 copy PC to LR, then branch
BXcd reg copy reg to PC
SWIcd imm24 software interrupt
LDRcdB reg, mem loads word/byte from memory
STRcdB reg, mem stores word/byte to memory
LDMcdum reg!, mreg loads into multiple registers
STMcdum reg!, mreg stores multiple registers
SWPcdB regd, regm, [regn] copies regm to memory at regn,old value at address regn to regd

Optional:
cd – Condition Code
s – Update flkag or not
b – byte or word instruction

ARM Instructions Add-Ons: Conditions

- Every instruction may have a condition appended:

Example:

MOV r1, r2 and EQ (zero flag set)

becomes

MOVEQ r1,r2

This means that the r2 will be moved to r1 only if the zero flag is set.

List of Conditions that Can be Added to Instructions

AL or omitted always
EQ equal (zero)
NE nonequal (nonzero)
CS carry set (same as HS)
CC carry clear (same as LO)
MI minus
PL positive or zero
VS overflow set
VC overflow clear
HS unsigned higher or same
LO unsigned lower
HI unsigned higher
LS unsigned lower or same
GE signed greater than or equal
LT signed less than
GT signed greater than
LE signed less than or equal

Example: Adding two numbers

- Implement the following program in assembler:

```
#include <stdio.h>


main()
{
  int a;
  int b;
  int c;


  a = 2;
  b = 3;
  c = b + c;

  printf("c=%d\n", c);
}
```

Example: Adding two numbers in Assembly using Registers

```
/* add-reg.s
   Adding two numbers using registers */
        .section        .rodata
printfArg:
        .ascii  "c=%d\n"

        /* Define variables
           int a; - r2
           int b; - r3
           int c; - r1
         */

        .text
```

Adding two numbers in Assembly using Registers (cont.)

```
        .global main                /* main() { */
main:
        push {r4-r9, fp, lr} /* Save pc and lr */

        mov     r2, #2          /* a=2; */
        mov     r3, #3          /* b=3; */
        add     r1, r2, r3      /* c = a + b; */

        ldr     r0, =printfArg
                  /* Load printf format in r0 */
                  /* second argument is in r1 */
                  /* r1 already has the result of a+b*/
        bl      printf   /* printf("c=%d\n", c); */

        pop {r4-r9, fp, pc} /* return from main */
                                /* } */
```

Adding two numbers in Assembly using Registers (cont.)

```
pi@raspberrypi:~/cs250/lab6-src$ gcc -o add-reg add-reg.s

pi@raspberrypi:~/cs250/lab6-src$ ./add-reg
c=5
```

Example: Adding Two Numbers Using Global Vars

```
/* add-global.s:
    Adding two numbers using global variables */

        .section        .rodata
printfArg:
        .ascii  "c=%d\n"

        .section .data

        .align 4

        /* Define variable 4 bytes each aligned to 4 bytes
           int a;
           int b;
           int c;
         */
        .comm   a,4,4
        .comm   b,4,4
        .comm   c,4,4
```

Adding Two Numbers Using Global Vars (cont.)

```
 .text

.global main                /* main() { */
main:
       push {r4-r9, fp, lr}            /* Save pc and lr */

        ldr     r3, =a                /* a = 2; */
        mov     r2, #2
        str     r2, [r3]

        ldr     r3, =b                /* b = 3; */
        mov     r2, #3
        str     r2, [r3]
```

Adding two numbers using Global Vars (cont.)

```
ldr     r2, =a           /* Read a and put it in r2 */
ldr     r2, [r2]

ldr     r3, =b           /* read b and put it in r3 */
ldr     r3, [r3]

add     r2, r2, r3           /* c = a + b; */
ldr     r3, =c
str     r2, [r3]

ldr     r0, =printfArg     /*Load printf format in r0 */
ldr     r1, =c
ldr     r1, [r1]             /* Load c in r1 */
bl      printf               /* printf("c=%d\n", c); */

pop {r4-r9, fp, pc} /* return from main */

                          /* } */
```

Adding two numbers using Global Vars (cont.)

```
pi@raspberrypi:~/cs250/lab6-src$ gcc -o add-global add-global.s
pi@raspberrypi:~/cs250/lab6-src$ ./add-global
c=5
```

Example: Read two numbers and add them

```
/* readadd.s
  Read two numbers and add them
pi@raspberrypi:~/cs250/lab6-src$ ./readadd
a: 8
b: 9
c=a+b=17
*/
        .section        .rodata
promptA:
        .ascii  "a: \000"
promptB:
        .ascii  "b: \000"
readA:
        .ascii  "%d\000"
readB:
        .ascii  "%d\000"
printC:
        .ascii  "c=a+b=%d\n\000"
```

Example: Read two numbers and add them (cont.)

```
.section .data
        .align 2

        /* Define variable 4 bytes each aligned to 4 bytes
           int a;
           int b;
         */
        .comm   a,4,4
        .comm   b,4,4
```

Example: Read two numbers and add them (cont.)

```
        .text
.global main                /* main() { */
main:
        push {r4-r9, fp, lr} /* Save pc and lr */

        ldr     r0, =promptA          /* printf(“a:”); */
        bl      printf

        ldr     r0, =readA            /* scanf(“%d ”, &a); */
        ldr     r1, =a
        bl      scanf

        ldr     r0, =promptB          /* printf(“b: “); */
        bl      printf

        ldr     r0, =readB            /* scanf(“%d ”, &b);*/
        ldr     r1, =b
        bl      scanf
                                      /* c = a + b;
        ldr     r0, =a                /* r0<- a */
        ldr     r0, [r0]
```

Example: Read two numbers and add them (cont.)

```
ldr     r1, =b              /* r1<- b */
ldr     r1, [r1]

add     r1, r0, r1             /* r1 <- r1 +r0*/

ldr     r0, =printC         /* printf("c=a+b=%d",c); */
bl      printf

pop {r4-r9, fp, pc} /* return from main */

                          /* } */
```

Example: Read two numbers and add them (cont.)

pi@raspberrypi:~/cs250/lab6-src$ gcc -o readadd readadd.s

pi@raspberrypi:~/cs250/lab6-src$ ./readadd

a: 7

b: 4

c=a+b=11

Mixing C and Assembly Language. Finding max in an array.

max.c:

```
#include <stdio.h>
#include <stdlib.h>
extern int maxarray(int *a, int n);
main()
{
	int n;
	int i;
	int * a;

	printf("How many elements in array? ");
	scanf("%d",&n);
	a = (int*) malloc(n*sizeof(int));

	for (i = 0; i < n; i++) {
		printf("a[%d]= ", i);
		scanf("%d", &a[i]);
	}

	int m = maxarray(a, n);
	printf("max=%d\n", m);
}
```

Mixing C and Assembly Language. Finding max in an array (cont.)

```
maxarray.s
/* Find maximum of an array of integers. Called from "C"
  extern int maxarray(int *a, int n);
*/
        .text

        .global maxarray        /* maxarray(int *a, int n) { */
                                         /* a: r0 */
                                         /* n: r1 */
maxarray:
        push {r4-r9, fp, lr}
                                /* Save pc, lr, r4, r5 */

        ldr     r2,[r0]                  /* max: r2  */
                                         /* max= a[0] */

        mov     r3,#0                    /* i: r3 */
                                         /* i=0; */
```

Mixing C and Assembly Language. Finding max in an array (cont.)

```
while:
        cmp     r3,r1        /* while (i!=n) { */
        beq     endmax

        mov     r4,r3        /* r4=a[i] */
        mov     r5,#4
        mul     r4,r4,r5
        add     r4,r0,r4 /* as r4=*(int*)((char*)a+4*i)*/
        ldr     r4,[r4]

        cmp     r2, r4      /* if (max < r4) max = r4 */
        bgt     nomax
        mov     r2,r4

nomax:
```

Mixing C and Assembly Language. Finding max in an array (cont.)

```
        mov     r5,#1       /* i++; */
        add     r3,r3,r5

        bal     while       /* Go back to while */
endmax:

        mov     r0,r2

        pop {r4-r9, fp, pc}
                          /* return from main */
                          return max;
                         /* } */
```

Mixing C and Assembly Language. Finding max in an array (cont.)

```
pi@raspberrypi:~/cs250/lab6-src$ gcc -o max
  max.c maxarray.s
pi@raspberrypi:~/cs250/lab6-src$ ./max
How many elements in array? 6
a[0]= 34
a[1]= 78
a[2]= 34
a[3]= 90
a[4]= 78
a[5]= 45
max=90
```

Implementing String Functions in ARM Assembly Language

- There are two functions to load/store bytes:
- ldrb reg1,[reg2]
  - Loads in reg1 the byte in address pointed by reg2
- strb reg1,[reg2]
  - Stores the least significant byte in reg1 byte in address pointed by reg2

Example: strcat function in ARM assembly

```
/* strcat-main.c:*/
#include <stdio.h>
#include <string.h>
extern char * mystrcat(char * a, char *b);

main()
{
        char s1[100];
        char s2[100];
        printf("s1? ");
        gets(s1);
        printf("s2? ");
        gets(s2);
        mystrcat(s1, s2);
        printf("s1+s2=%s\n", s1);
}

// Implemented in Assembly Language in mystrcat.s
// Shown here to teach you the algorithm used.
// char * mystrcat(char * a, char *b) {
//        while (*a) a++;
//        while (*b) { *a=*b; a++; b++;}
//        *a=0;
// }
```

Example: strcat function in ARM assembly (cont.)

```
/* Concat two strings a, b. Result is in a.
  extern char * mystrcat(char *a, char *b);
*/
        .text

        .global mystrcat
                                        /* a: r0 */
                                        /* b: r1 */
mystrcat:
        push {r4-r9, fp, lr}            /* Save pc, lr, r4*/

                                        /* Skip chars in a */
skip:
        ldrb    r2,[r0]                 /* r2 <- *a */
        mov     r3,#0
        cmp     r2,r3
        beq     endskip                 /* if (*a == 0) jump endskip */
        mov     r3,#1
        add     r0,r0,r3                /* a++ */
        bal     skip                    /* go to skip */

endskip:
```

Example: strcat function in ARM assembly (cont.)

```
skip2:                                        /* Add char by char *b to *a until we
   find the end of *b */
        ldrb    r4,[r1]                      /* r4 <- *b */
        mov     r3,#0
        cmp     r4,r3
        beq     endcat                       /* if (*b == 0) jump endcat */
        strb    r4,[r0]                      /* *a = *b; */
        mov     r3,#1
        add     r0,r0,r3                     /* a++ */
        add     r1,r1,r3                     /* b++ */
        bal     skip2                        /* go to skip2 */

endcat:
        mov     r3, #0                       /* *a = 0; */
        strb    r3, [r0]

        pop {r4-r9, fp, pc} /* return from mystrcat */
```

Example: strcat function in ARM assembly (cont.)

```
pi@raspberrypi:~/cs250/lab6-src$ gcc -o strcat-main strcat-main.c
   mystrcat.s
pi@raspberrypi:~/cs250/lab6-src$ ./strcat-main
s1? Hello
s2? World
s1+s2=HelloWorld
pi@raspberrypi:~/cs250/lab6-src$
```

Using gdb to debug your code

- **`pi@raspberrypi:~/cs250/lab6-src$ gcc -g -o strcat-main strcat-main.c mystrcat.s`**

```
gdb strcat-main
gdb>break main
gdb>break mystrcat
gdb>run
gdb>cont
gdb>disassemble
gdb>nexti
gdb>stepi
gdb>print $r0
```

PIC 18 Introduction

PIC18

- In the labs you will use the PIC18
- This is a 8 bit processor that provides
  - Digital I/O
  - Analog to Digital Conversion
  - Pulse Width Modulation
  - USB support
  - RS232 (Serial Line)
- Data Sheet of PIC18:
  - http://ww1.microchip.com/downloads/en/DeviceDoc/39632e.pdf

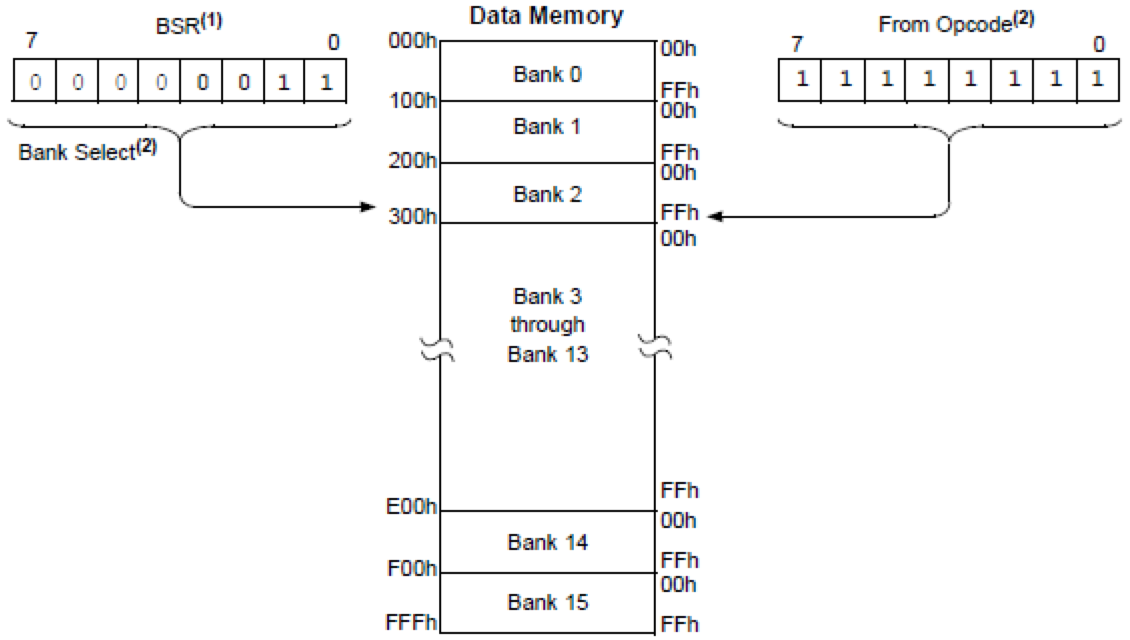
PIC18

- It follows a **Harvard Architecture**, that is, code and data are stored in separate memory.
  - Code - 32KB
  - Data - 4KB
- Instructions can be 2 or 4 byte long.
- The data word is 1 byte.

Data Memory

- RAM is 4KB or 2^{12}
- Therefore, pointers are 12 bits long
- The memory is divided into 16 banks.
- Each bank is 256 bytes long.
- That is 16x256=4KB

FIGURE 5-6: USE OF THE BANK SELECT REGISTER (DIRECT ADDRESSING)



Note 1: The Access RAM bit of the instruction can be used to force an override of the selected bank (BSR<3:0>) to the registers of the Access Bank.

2: The `MOVFF` instruction embeds the entire 12-bit address in the instruction.

Memory Addresses

- The instructions that access data use a reduced pointer that is 8 bits long (0 to 255)
- The remaining 4 highest bits are specified by the argument “a” in each instruction.
  - If a=0 the address refers to the “Access Bank” that uses bank 0 for 0x00 to 0x5F and 0x60 to 0xFF from bank 15.
  - If a=1, the 4 highest bits are contained in a register called BSR (Bank Selection Register)
  - 99% of the time a=0 in your programs.

Special Function Registers and General Function Registers

- The data memory is divided into
  - SFRs – Special Function Registers. Used for control and status of the processor.
  - GPRs – General Purpose Registers. Used to store temporal results in user application.

TABLE 5-1: SPECIAL FUNCTION REGISTER MAP

Address	Name	Address	Name	Address	Name	Address	Name	Address	Name
FFFh	TOSU	FDFh	INDF2(1)	FBFh	CCPR1H	F9Fh	IPR1	F7Fh	UEP15
FFEh	TOSH	FDEh	POSTINC2(1)	FBEh	CCPR1L	F9Eh	PIR1	F7Eh	UEP14
FFDh	TOSL	FDDh	POSTDEC2(1)	FBDh	CCP1CON	F9Dh	PIE1	F7Dh	UEP13
FFCh	STKPTR	FDCh	PREINC2(1)	FBCh	CCPR2H	F9Ch	—(2)	F7Ch	UEP12
FFBh	PCLATU	FDBh	PLUSW2(1)	FBBh	CCPR2L	F9Bh	OSCTUNE	F7Bh	UEP11
FFAh	PCLATH	FDAh	FSR2H	FBAh	CCP2CON	F9Ah	—(2)	F7Ah	UEP10
FF9h	PCL	FD9h	FSR2L	FB9h	—(2)	F99h	—(2)	F79h	UEP9
FF8h	TBLPTRU	FD8h	STATUS	FB8h	BAUDCON	F98h	—(2)	F78h	UEP8
FF7h	TBLPTRH	FD7h	TMR0H	FB7h	ECCP1DEL	F97h	—(2)	F77h	UEP7
FF6h	TBLPTRL	FD6h	TMR0L	FB6h	ECCP1AS	F96h	TRISE(3)	F76h	UEP6
FF5h	TABLAT	FD5h	T0CON	FB5h	CVRCON	F95h	TRISD(3)	F75h	UEP5
FF4h	PRODH	FD4h	—(2)	FB4h	CMCON	F94h	TRISC	F74h	UEP4
FF3h	PRODL	FD3h	OSCCON	FB3h	TMR3H	F93h	TRISB	F73h	UEP3
FF2h	INTCON	FD2h	HLVDCON	FB2h	TMR3L	F92h	TRISA	F72h	UEP2
FF1h	INTCON2	FD1h	WDTCON	FB1h	T3CON	F91h	—(2)	F71h	UEP1
FF0h	INTCON3	FD0h	RCON	FB0h	SPBRGH	F90h	—(2)	F70h	UEP0
FEFh	INDF0(1)	FCFh	TMR1H	FAFh	SPBRG	F8Fh	—(2)	F6Fh	UCFG
FEEh	POSTINC0(1)	FCEh	TMR1L	FAEh	RCREG	F8Eh	—(2)	F6Eh	UADDR
FEDh	POSTDEC0(1)	FCDh	T1CON	FADh	TXREG	F8Dh	LATE(3)	F6Dh	UCON
FECh	PREINC0(1)	FCCh	TMR2	FACh	TXSTA	F8Ch	LATD(3)	F6Ch	USTAT
FEBh	PLUSW0(1)	FCBh	PR2	FABh	RCSTA	F8Bh	LATC	F6Bh	UEIE
FEAh	FSR0H	FCAh	T2CON	FAAh	—(2)	F8Ah	LATB	F6Ah	UEIR
FE9h	FSR0L	FC9h	SSPBUF	FA9h	EEADR	F89h	LATA	F69h	UIE
FE8h	WREG	FC8h	SSPADD	FA8h	EEDATA	F88h	—(2)	F68h	UIR
FE7h	INDF1(1)	FC7h	SSPSTAT	FA7h	EECON2(1)	F87h	—(2)	F67h	UFRMH
FE6h	POSTINC1(1)	FC6h	SSPCON1	FA6h	EECON1	F86h	—(2)	F66h	UFRML
FE5h	POSTDEC1(1)	FC5h	SSPCON2	FA5h	—(2)	F85h	—(2)	F65h	SPPCON(3)
FE4h	PREINC1(1)	FC4h	ADRESH	FA4h	—(2)	F84h	PORTE	F64h	SPPEPS(3)
FE3h	PLUSW1(1)	FC3h	ADRESL	FA3h	—(2)	F83h	PORTD(3)	F63h	SPPCFG(3)
FE2h	FSR1H	FC2h	ADCON0	FA2h	IPR2	F82h	PORTC	F62h	SPPDATA(3)
FE1h	FSR1L	FC1h	ADCON1	FA1h	PIR2	F81h	PORTB	F61h	—(2)
FE0h	BSR	FC0h	ADCON2	FA0h	PIE2	F80h	PORTA	F60h	—(2)

Working Register (WREG)

- Most arithmetic and logical operations use a register called *Working Register* or *WREG*.

Processor Status Register (PSR)

- This is a register that contains the status of the Arithmetic Logical Unit.
- It is separated in bits
  - N – Negative bit. Turns to 1 if the result of the last operation was negative (highest bit is 1).
  - OV – Overflow bit. Last operation in ALU results in an overflow.
  - Z – Zero bit. Last operation in ALU resulted in 0.
  - C – Carry or Borrow. Set to 1 if addition resulted in carry or borrow.
- Also the PSR is used in multiple branch instructions.

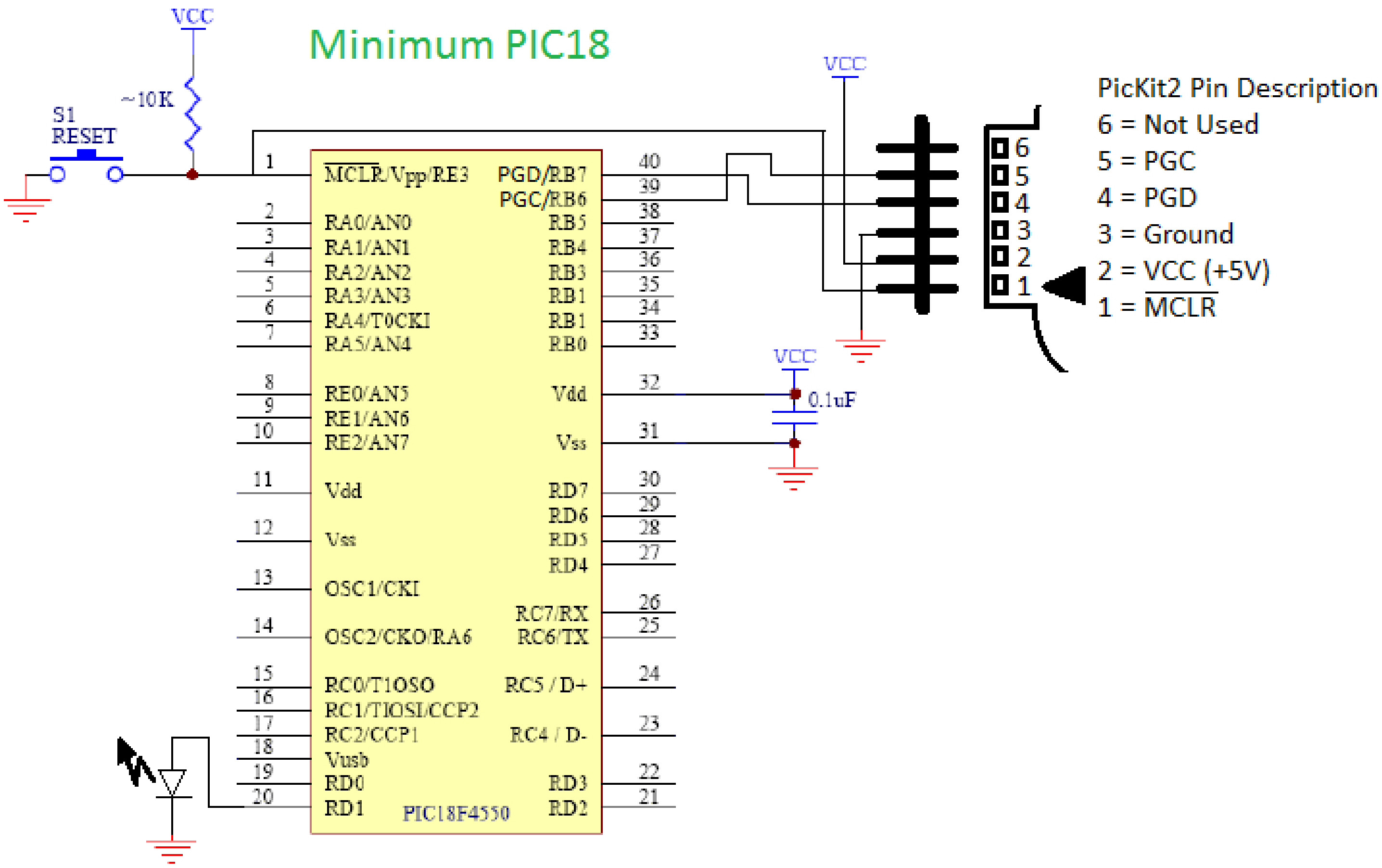
Digital Input/Output

- PORTA, PORTB, PORTC, PORTD
  - They are the registers that are mapped to the inputs/outputs of the PIC18.
  - Each bit in the port is identified as RA0, RA1 …RA7, RB1, RB2…RB7 and so on,
- TRISA, TRISB, TRISC, TRISD
  - Used to configure ports as input/output.
  - Each bit can be configured to be a digital input or output..
    - 0 – Output
    - 1 – Input

Digital Input/Output

- When configured PORTA as output for example
  - 0 in bit RA0 of PORTA gives 0V in terminal RA0
  - 1 in bit RA0 of PORTA gives +5V in terminal RA0
- When configured PORTA as input,
  - 0V in terminal RA0 can be read as 0 in bit RA0 of PORTA
  - +5V in terminal RA0 can be read as 1 in bit RA0 of PORTA

Minimum PIC18



Addressing Modes

- Inherent (Immediate)
  - Used in instructions that do not need an argument such as SLEEP and RESET
- Literal
  - Used in instructions that specify a numeric constant such as “MOVLW 0x40” that loads 0x40 in WREG
- Direct
  - Used in instructions that need an address as argument such as “MOVWF 0x080” that moves WREG into 080.
- Indirect
  - A register or memory location contains the address of the source or destination.

Indirect Addressing

- It uses the FSR registers and the INDF operand.
- There are four registers:
  - FSR0, FSR1, FSR2, FSR3, and the corresponding
  - INDF0, INDF1, INDF2, INDF3.
- INDF0 to INDF3 are “virtual registers”.
- A read from INDF2 for example, reads the register at the address stored in FSR2.
- Since FSRs is 12 registers long, you can use FSRL(lower byte) and FSRH(higher 4 bits) for the instructions.

Byte Operations

d = 0 means destination is WREG.

d = 1 means destination is a file register and it is the default.

a is the access bank. By default it is 0.

- ADDWF f,d,a - Add W to f where d=0->W, d=1->f, a is generally not specified (access bank stuff)
- ADDWFC f,d,a - Add W and Carry bit to f
- ANDWF f,d,a - And W with f
- CLRF f,a Clear f
- COMF f,a Complement f
- CPFSEQ Compare, skip if f==W
- CPFSGT Compare, skip if f > W
- CPFSLT Compare, skip if f < W

Byte Operations (cont.)

- DECF f,d,a Decrement f
- DECFSZ f,d,a Dec f, skip if 0
- DCFSNZ f,d,a Dec f, skip if not 0
- INCF f,d,a Increment f
- INCFSZ f,d,a Increment f, skip if zero
- INFSNZ f,d,a Increment f, skip if not zero
- IORWF f inclusive-OR W with f
- MOVF f,d,a Move f (usually to W)
- MOVFF f,ff Move f to ff
- MOVWF f,a Move W to f
- MULWF f,a W x f

Byte Operations (cont.)

- NEGF f,a Negate f
- RLCF f,d,a Rotate left f thru Carry (not-quite multiply by 2 with carry)
- RLNCF f,d,a Rotate left (no carry)
- RRCF f,d,a Rotate right through Carry
- RRNCF f,d,a Rotate right f (no carry)
- SETF f,a Set f = 0xff
- SUBFWB f,d,a Subtract f from w with Borrow
- SUBWF f,d,a Subtract W from f
- SUBWFB f,d,a Subtract W from f with Borrow
- SWAPF f,d,a Swap nibbles of f
- XORWF f,d,a W XOR f

Bit Operations (cont.)

- BCF f,b,a Bit clear, bit is indexed 0 to 7
- BSF f,b,a Bit set
- BTFSC f,b,a Bit test, skip if clear
- BTFSS f,b,a Bit test, skip if set
- BTG f,b,a Bit toggle

Control Operations (cont.)

- BC n Branch if Carry, n is either a relative or a direct address
- BN n Branch if Negative
- BNC n Branch if Not Carry
- BNN n Branch if Not Negative
- BNOV n Branch if Not Overflow
- BNZ n Branch if Not Zero
- BOV n Branch if Overflow
- BRA n Branch Unconditionally
- BZ n Branch if Zero CALL n, s Call Subroutine

Control Operations (cont.)

- CLRWDT Clear Watchdog Timer
- DAW Decimal Adjust W
- GOTO n Go to address
- NOP No operation
- POP Pop top of return stack (TOS)
- PUSH Push top of return stack (TOS)
- RCALL n Relative Call
- RESET Software device reset
- RETFIE Return from Interrupt and Enable Interrupts
- RETURN s Return from subroutine
- SLEEP Enter SLEEP Mode

Operations with Literals (constants)

- ADDLW kk Add literal to W
- ANDLW kk And literal with W
- IORLW kk Incl-OR literal with W
- LFSR r,kk Move literal (12 bit) 2nd word to FSRr 1st word
- MOVLB k Move literal to BSR<3:0>
- MOVLW kk Move literal to W
- MULLW kk Multiply literal with W
- RETLW kk Return with literal in W
- SUBLW kk Subtract W from literal
- XORLW kk XOR literal with W

Common PIC Assembler Constructions

- Including the PIC18 constant defined values
  - Add

    #include “P18f4550.INC”

    at the beginning of the file
  - In this way you can specify PORTC instead of 0xF82 when specifying names of registers

Defining a variable

- To define space for a variable use “res”.

```
Delay1 res 2
```

- This defines a variable called Delay1 that will take 2 bytes.
- Make sure that it is at the beginning of the line.

Using registers

- Loading a constant into WREG

```
MOVLW 0x40
```

- Moving the value from a register to WREG

```
MOVF reg,0
```

- Moving the value of WREG into a register

```
MOVWF reg
```

- Moving the value of a register reg1 to reg2

```
MOVFF reg1, reg2
```

Adding and Subtracting

- Add reg1 and reg2. Put result in reg1

```
MOVF reg1,0 ; WREG = reg1
ADDWF reg2,0; WREG = WREG + reg2
MOVWF reg1  ; reg1 = WREG
```

- Subtract reg2 - reg1. Put result in reg2

```
MOVF reg1,0 ; WREG = reg1
SUBWF reg2,0; WREG = reg2-WREG
MOVWF reg2  ; reg2 = WREG
```

Subroutines

- To call a subroutine

```
    ...
    CALL foo ; Calling subroutine foo
    ...
    ...
```

- To define a subroutine

```
foo   ; Defintion of foo
    ...
    RETURN ; Return from subroutine
```

If/else statements

- If (reg1 == 0x40) {XXX} else { YYY}

```
  MOVLW 0x40; WREG = reg1
  CPFSEQ reg1
  GOTO elsepart
  ….; XXX Then part
  GOTO endifpart
elsepart
  … ; YYY else part
endifpart
```

Using Arrays

- Arrays are implemented using Indirect Indexing
- Defining an array of bytes called “myArray” of 4 elements:

```
myArray res 4
```

- Initializing array:

```
MOVLW 0xFE              ; myArray[0]=0xFE
MOVWF myArray
MOVLW 0xFD              ; myArray[1]=0xFD
MOVWF myArray +1
MOVLW 0xFB              ; myArray[2]=0xFB
MOVWF myArray +2
MOVLW 0xF7              ; myArray[3]=0xF7
MOVWF myArray +3;
```

Using Arrays

- Indexing the Array myArray[i].
- Address is stored in FSR0 and then it is dereferenced from INDF0

```
LFSR 0, myArray ; Load array address in FSR0
MOVF i,0        ; Load the value of i into WREG
ADDWF FSR0L,1   ; Add myArray and i to get address
                 ; of ith element.
MOVF INDF0,0    ; The ith element can be read
                 ; from INDF0. Read it and put
                 ; it into WREG. WREG=myArray[i]
MOVWF PORTB     ; Now do something with it like
                 ; writing it to PORTB
```

Simple Program. LED Blink

```
#include "P18f4550.INC"

    CONFIG WDT=OFF; disable watchdog timer
    CONFIG MCLRE = ON; MCLEAR Pin on
    CONFIG DEBUG = ON; Enable Debug Mode
    CONFIG LVP = OFF; Low-Voltage programming disabled (necessary for debugging)
    CONFIG FOSC = INTOSCIO_EC;Internal oscillator, port function on RA6

    org 0; start code at 0


Delay1 res 2 ;reserve space for the variable Delay1
Delay2 res 2 ;reserve space for the variable Delay2


Start:
    CLRF PORTD ; Clear all D outputs
    CLRF TRISD ; Make output all the bits in D
    CLRF Delay1 ; Initialize both counters with 0s.
    CLRF Delay2


MainLoop:
    BTG PORTD,RD1 ;Toggle PORT D PIN 1 (20)


Delay:
    DECFSZ Delay1,1 ;Decrement Delay1 by 1, skip next instruction if Delay1 is 0
           ;Delay1 will be decremented 256 times before skipping
    GOTO Delay
    DECFSZ Delay2,1 ;Decrement Delay2 by 1, skip next instruction if Delay2 is 0
           ;Delay1 will be decremented 256 times before skipping.
```

Another Example. Rotate Segments

```
#include "P18f4550.INC"

        CONFIG WDT=OFF; disable watchdog timer
        CONFIG MCLRE = ON; MCLEAR Pin on
        CONFIG DEBUG = ON; Enable Debug Mode
        CONFIG LVP = OFF; Low-Voltage programming disabled (necessary for debugging)
        CONFIG FOSC = INTOSCIO_EC;Internal oscillator, port function on RA6

        org 0; start code at 0



Delay1 res 2 ; variable Delay1
Delay2 res 2 ; variable Delay2
Delay3 res 2 ; variable Delay3



Start:
        CLRF PORTD ; Initialize with 0's output D.
        CLRF TRISD ; Make port D output
        CLRF Delay1; Clear delay variables
        CLRF Delay2

        SETF TRISC ; Make port c an input

        MOVLW H'40' ; Initialize delay3 to 0x40. This is the delay used to rotate the segments.
        MOVWF Delay3

        BSF PORTD,RD0 ;Turn on bit 0 in RD0
```

Another Example (cont.)

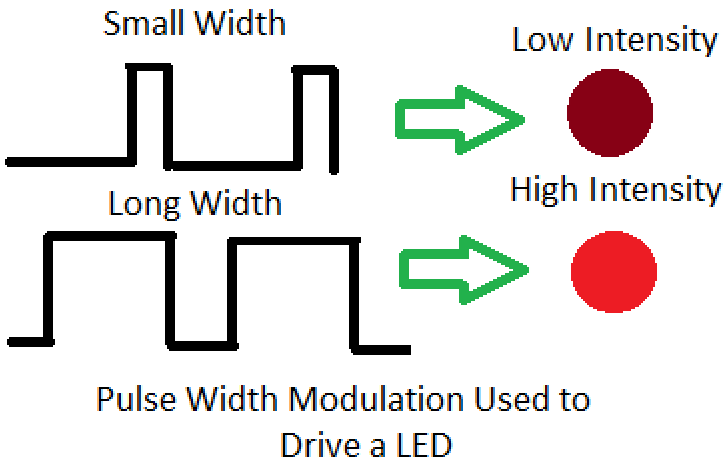
```
MainLoop:
    RRNCF PORTD ; Rotate bits in D. This causes the segments in display to shift.

    MOVF Delay3,0 ; Reload Delay2 eith the value of Delay3. Delay2 controls the rate the
    MOVWF Delay2 ; rotate takes place.

    MOVLW H'F0' ; Test if Delay3 is at the maximum of 0xF0 or more. If that is the case, do not
    CPFSLT Delay3 ; read the left switch.
    goto noincrement

    MOVLW 4 ; Read the left switch.
    BTFSS PORTC,0 ; If the switch is 0 (gnd), then increase Delay3 by 4, otherwise skip the increment.
    ADDWF Delay3,1


noincrement:

    MOVLW H'05' ; Test if Delay3 is at the minimum pf 0x5 or less. If that is the case do not
    CPFSGT Delay3 ; read the right switch.
    goto Delay

    MOVLW 4 ; Read the right switch.
    BTFSS PORTC,1 ; If the switch is 0, then decrement Delay3 by 4, otherwise skip the decrement
    operation.
    SUBWF Delay3,1


Delay:

    DECFSZ Delay1,1 ;Decrement Delay1 by 1, skip next instruction if Delay1 is 0
    GOTO Delay
    DECFSZ Delay2,1 ;Decrement Delay1 by 1, skip next instruction if Delay1 is 0
    GOTO Delay
    GOTO MainLoop

    end
```

Example: Driving a Full-Color LED

- To drive the full-color LED you will use Pulse Width Modulation (PWM).
- PWM sends pulses to the LED with different widths to the three color LEDs.
- If for example, the width of the pulse is small for the red LED, then the red LED will display a low intensity red light.
- If the red LED receives a pulse with a wide width, then the red LED will display a high intensity red light.

Pulse Width Modulation

- Short Width = Low Intensity
- Long Width = High Intensity



Pulse Width Modulation Example

```
 MOVFF maxColor, redCount
 MOVFF maxColor, greenCount
 MOVFF maxColor, blueCount



MainLoop:
    ;;;;;; RED LED ;;;;;;
    ; Decrement redCount
    DECFSZ    redCount,1
    GOTO afterDecRedCount

    ; if redCount reaches 0 turnoff red led
    BSF PORTC,RC0
    ; restart redCount with 255
    SETF redCount

afterDecRedCount
    ; if redCount == red turn on red led.
    MOVF redCount,0
    CPFSEQ red
    GOTO updateGreen
    BCF PORTC, RC0
…

; Same for green and blue
goto MainLoop
```

Lab5 Driving a Full Color LED Algorithm

- Examples are given that shows you how to drive the full color led and how to display the Hello message in the display.
- Read them and understand them.
- They will be used as the base for your project

Algorithm for Driving Full Color LED

- Start
  - Initialize Ports and Registers
  - Initialize colors and counters
- MainLoop
  - Put in a variable ***val*** the current color value (red, green, blue)
  - Read button 1 and 2. If they are “on” increase or decrease ***val***. Make sure that val is not increased beyond maxColor and is not decreased beyond 0.
  - Update “***msg***” (the display buffer) with:
    - msg[0]= c[currentColor]
      - where c is an array with the characters “r”, “g” or b” in seven-segment values.
    - msg[1]= “=“
      - in seven segment value “=“ is(0x48)
    - msg[2] = digit[(val>>4)&0xFF]
      - Displays most significant nibble of val
      - digit is an array with the hex digits in seven segment value.
    - msg[3]=digit[val&0xFF]
      - Displays least significant nibble of val

Algorithm for Driving Full Color LED (cont.)

- Store val in currentColor red, green or blue
- Update Display. See example code.
- Read button 3 to change color if necessary. Use a variable *previouslyPressed* to store the previous status of the button.
- Only update the color name if previouslyPressed is false and button3 is pressed.
- To update the color name write into msg (the display buffer" the name of the color in seven-segment values.
- Now refresh the red, green, blue LEDs PWM See example code.

- Goto MainLoop