# ECEN 449 – Microprocessor System Design


## Review of C Programming

# Objectives of this Lecture Unit

- Review C programming basics
  - Refresh programming skills

# Basic C program structure

```
# include <stdio.h>
main()
{
  printf ("Hello world\n");
}
```

- Compilation of a program:
  gcc –o hello hello.c --- produces hello as the executable file
  --o specifies the name of the executable file
- Other flags:
  - Wall – turn on all warnings, useful for debugging purposes
  - o2 – turn optimization on
  - ansi –use ANSI C

# Basic Program Structure

- C programs consist of at least one main() function
  - Can have other functions as needed
- Functions can return values or return nothing
  - void function1 (int arg1, float arg2)
    - Takes an integer as first argument, float as second input
    - Returns no output
  - void function1 (arg1, arg2)
    - int arg1, float arg2;
    - Alternate description of the same function
  - int function2 ()
    - Takes no input, but returns an integer

# Libraries

- C employs many libraries
  - Make it easy to use already written functions in standard libraries
  - stdio.h –standard io –used to input/output data to/from the program
  - Other libraries such as math.h etc exist…

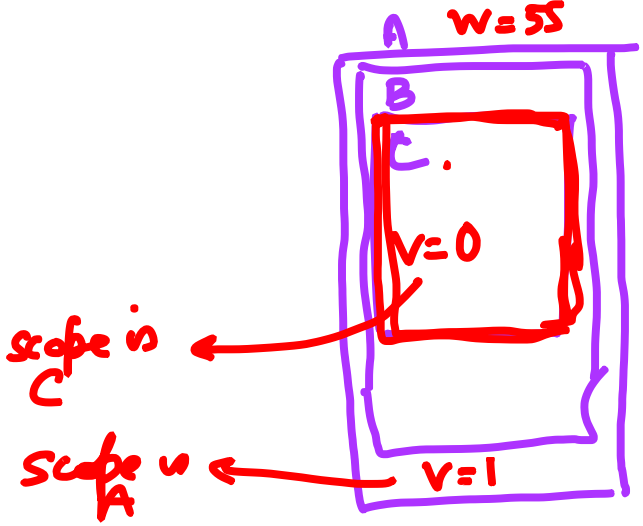
# Data types

- Int
  - Default size integer
  - Can vary in number of bits, but now mostly 32-bits
- Short
  - Used normally for 16-bit integers
- Long
  - Normally 32 bit integers, increasingly 64 bits
- Unsigned
  - The 32 bits are used for positive numbers, $0 \ldots 2^{32} - 1$
- Float
  - IEEE standard 32-bit floating point number
- Double
  - Double precision IEEE floating point number, 64 bits
- Char
  - Used normally for 8-bit characters

# Operators

- Assignment A = B;
  - Makes A equal to B
- A-B, A+B, A*B have usual meanings
- A/B depends on the data types
  - Actual division if A is float or double
  - Integer division if A and B are integers (ignore the remainder)
- A%B = remainder of the division
- (A == B) → output is predicate
  - Compares A and B and returns true if A is equal to B else false
- A++ --increment A i.e., A = A+1;
- A-- decrement A i.e., A = A – 1;

# Local vs. Global variables --Scope




- Declared variables have "scope"
- Scope defines where the declared variables are visible
- Global variables
  - Defined outside any function
  - Generally placed at the top of the file after include statements
  - Visible everywhere
  - Any function can read & modify them
- Local variables
  - Visible only within the functions where declared
  - Allows more control

✗ for debug

✓ for debug

# Global vs. Local variables

- Safer to use local variables
  - Easier to figure out who is modifying them and debug when problems arise
  - Pass information to other functions through values when calling other functions

```
function1 ()
{
   int arg1, arg2;              /* local to function1 */
   float arg3;                  /* local to function1 */
   arg3 = function2 (arg1, arg2);
}
```

# Global vs. local variables

```
int glob_var1, glob_var2;   /* global variables, declared out of scope for all functions
    */

main ()
{
   glob_var1 = 10;
   glob_var2 = 20;
….
}
function1()
{
   glob_var1 = 30;
}
```

# define

substituted inline

- # define REDDY-PHONE-NUM 8457598
  - Just like a variable, but can't be changed
  - Visible to everyone
- # define SUM(A, B) A+B
  - Where ever SUM is used it is replaced with +
  - For example SUM(3,4) = 3+4
  - Can be easier to read
  - Macro function
  - This is different from a function call
- #define string1 "Aggies Rule"

# Loops

- for (i=0; i< 100; i++)
  a[i] = i;

  Another way of doing the same thing:

```
i =0;
while (i <100)
{
   a[i] = i;
   i++;
}
```

# Arrays & Pointers

- int A[100];
- Declares A as an array of 100 integers from A[0], A[1]…A[99].
- If A[0] is at location 4000 in memory, A[1] is at 4004, A[2] is at 4008 and so on
  - When int is 4 bytes long
  - Data is stored consecutively in memory
- This gives an alternate way of accessing arrays through pointers
- Pointer is the address where a variable is located
- &A[0] indicates the address of A[0]

# Arrays and pointers

- int *A;
- This declares that A is a pointer and an integer can be accessed through this pointer
- There is no space allocated at this pointer i.e., no memory for the array.
- A = (int *) malloc(100 * sizeof(int));
- malloc allocates memory space of 100 locations, each the size that can hold an int.
- Just to be safe, we cast the pointer to be of integer type by using (int *)
- Now *A gives the first value of the array *(A+4) gives the second value…so on…

# Two ways to do arrays in C.

1. int A[100]

- an 'int' is 32b
- a byte is 8b
- so an 'int' is 4 bytes (or 4B)

- the above array needs 400B

- memory is addressed in bytes. So the array may be stored in locations:

  A[0] - 4000
  A[1] - 4004
  A[2] - 4008

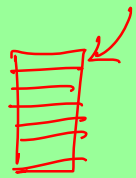
- Earlier languages hid the memory locations from users
- C exposes memory location to the user
  *(addr) ≡ "contents of" addr (pointer)
  &(value) ≡ "addr of" value
  &(A[1]) = 4004

13.2

2 `int *A  // A is pointer to ints`

`A = (int *) malloc (100 * sizeof (int))`

- `(int *)` → cast to int
- `malloc` → returns addr of contiguous memory chunk (say 4000)
- `sizeof (int)` → in bytes



`free (A) // after done`

`*(A)`


malloc hands out address chunks to users

→ users should be polite

- → not write outside allocation (random crashes)
- → free the memory (or system can run out!)

"out of memory allocating 12 bytes"

# Pointers

- Pointers are very general
- They can point to integers, floats, chars, arrays, structures, functions etc..
- Use the following to be safe:
- int *A, *B; float *C;

```
A = malloc (100 * sizeof(int));
B = (int *) (A + 1);  /* B points to A[1] */
```

Casting (int *) is safer even though not needed here…

```
C = (float *) A[1];  /* what does this say? */
```

# Structures

- A lot of times we want to group a number of things together.
- Use struct..
- Say we want a record of each student for this class
- Struct student {

```
        string name[30];
        int uin;
        int score_hw[4];
        int score_exam[2];
        float score_lab[5];
      };
Struct student 449_students[50];
```

# Structures

- 449_students[0] is the structure holding the first students' information.
- Individual elements can be accessed by 449_students[0].name, 449_students[0].uin…etc..
- If you have a pointer to the structure use 449_studentsp->name, 449_studentsp->uin etc…

# I/O

- getchar()
- putchar()
- FILE *fptr;
- fptr = fopen(filename, w);

  for (i= 0; i< 100; i++) fprintf(fptr, “%d\n”, i);

  fclose(fptr);
- fscanf(fptr, “%d\n”, i);

# Function calls and control flow

- When a function is called –a number of things happen
- Return address is saved on the stack
- Local variables saved on the stack
- Program counter loaded with called function address
- Function call executed
- On return, stack is popped to get return address, local variables
- Execution starts from the returned address

# Open()

- Library:

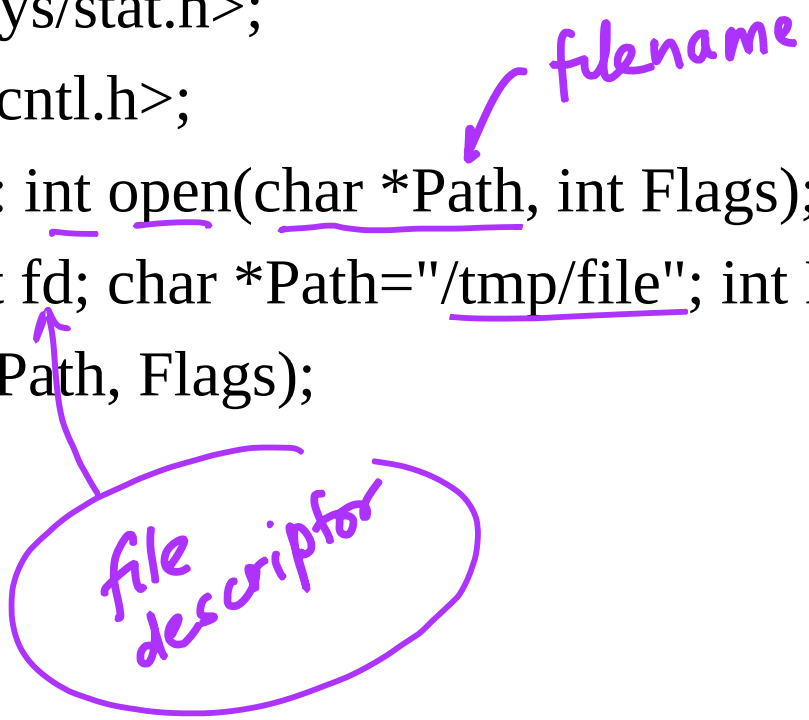
  include <sys/types.h>;

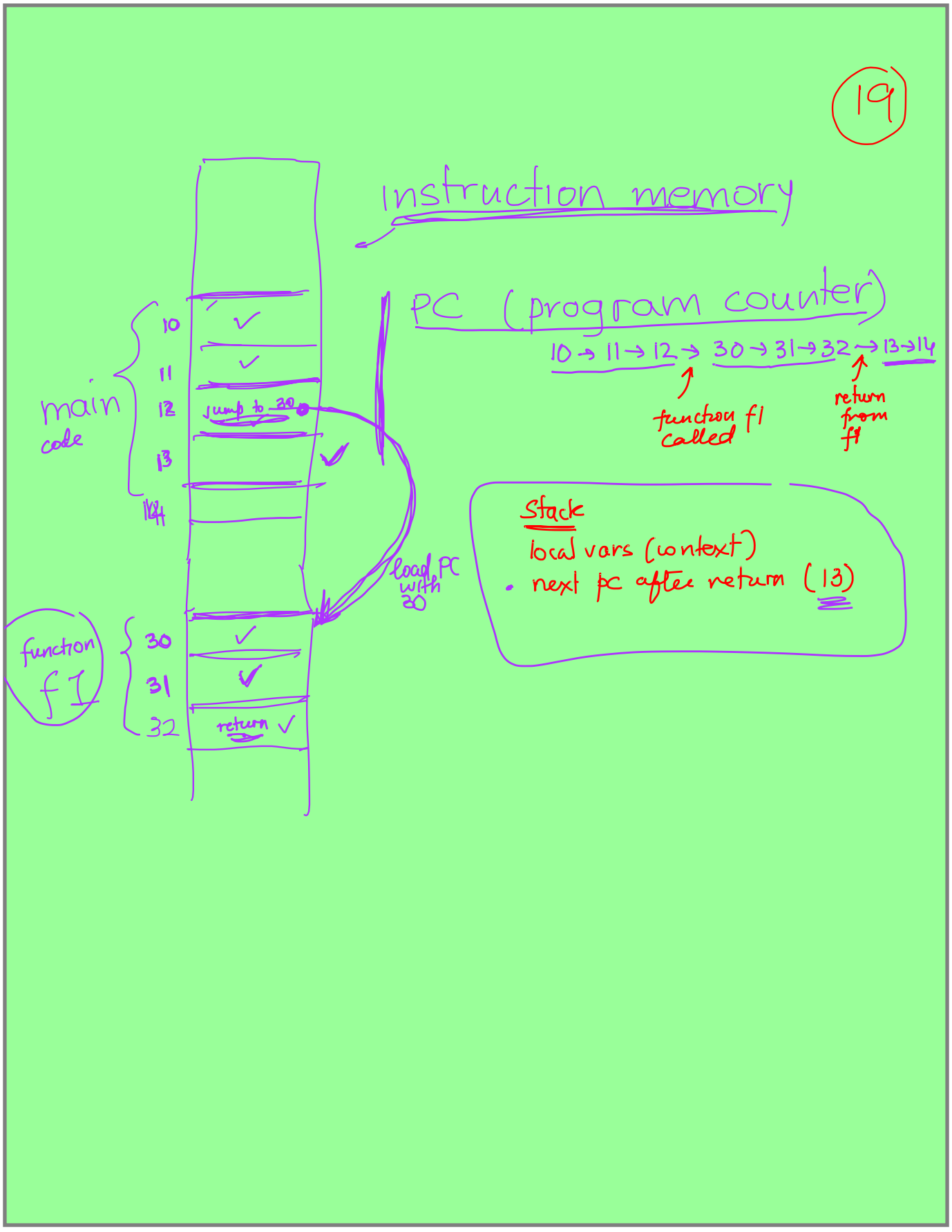
  include <sys/stat.h>;

  include <fcntl.h>;

- Prototype: int open(char *Path, int Flags);
- Syntax: int fd; char *Path="/tmp/file"; int Flags= O_WRONLY;

  fd = open(Path, Flags);

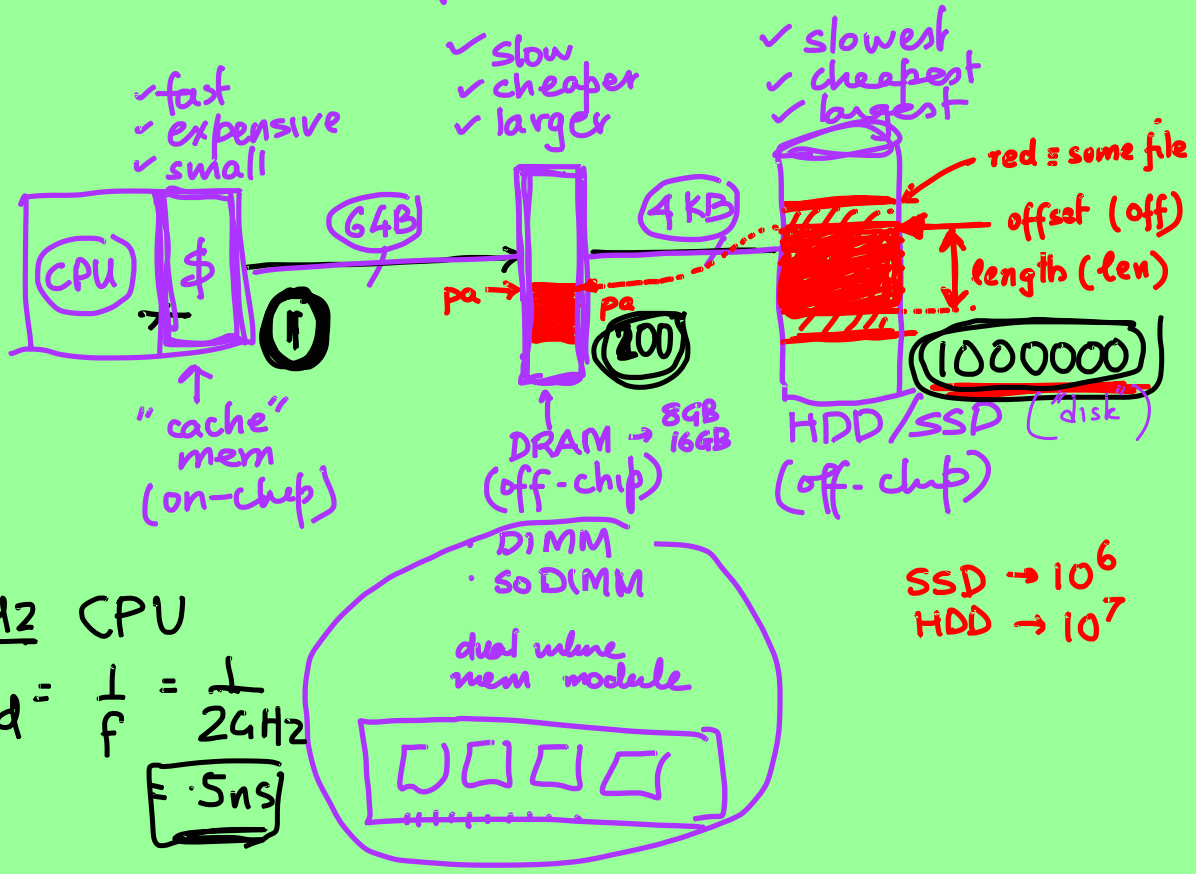



19
instruction memory
PC (program counter)
10 → 11 → 12 → 30 → 31 → 32 → 13 → 14
function f1 called
return from f1
main code
10
11
12
jump to 30
13
14
load PC with 30
function f1
30
31
32
return
Stack
local vars (context)
next pc after return (13)

21

# Memory hierarchy in a computer

→ I want fast memory, lots of it too!
→ But fast memory is expensive
  - slow memory is cheap
- So we improvise!

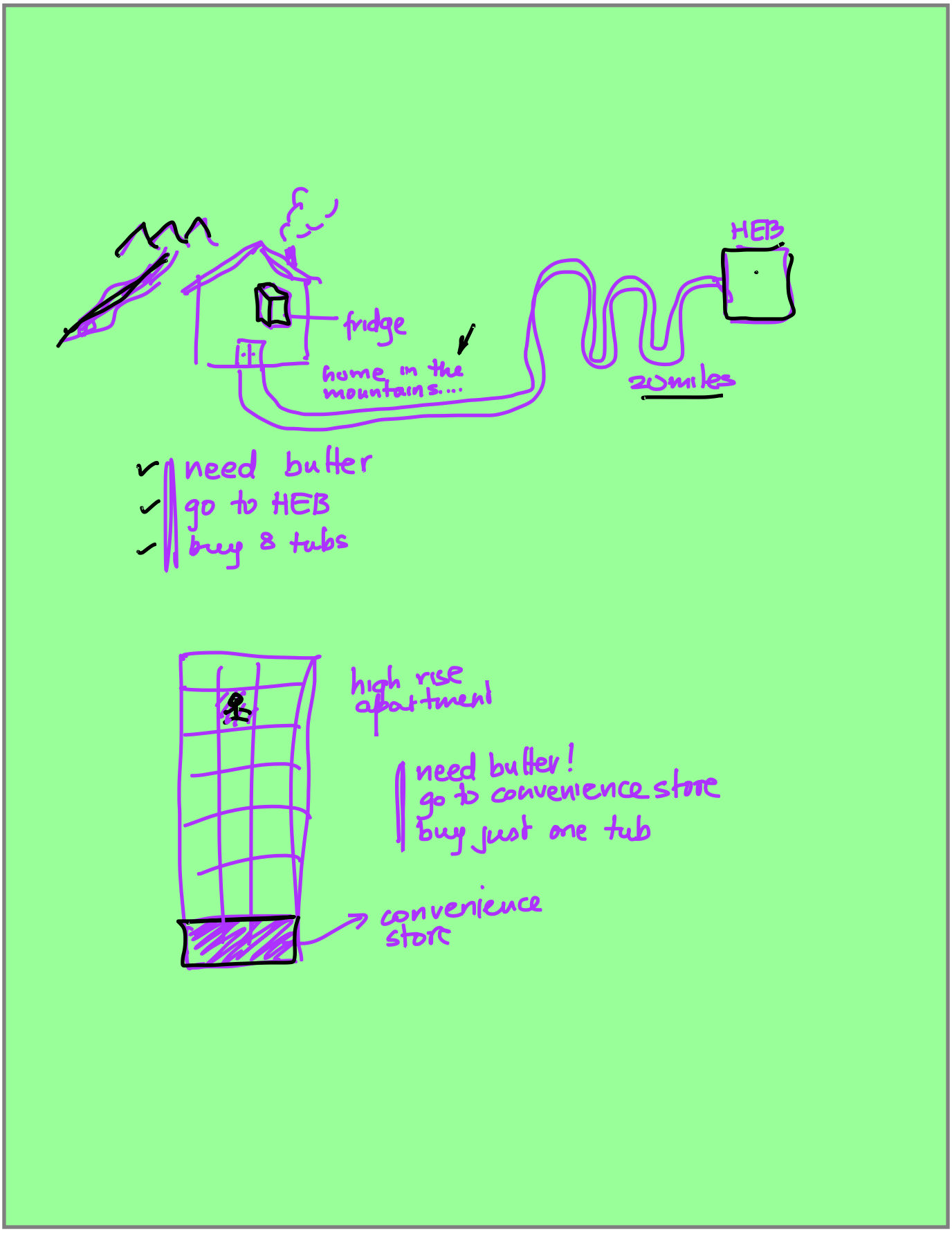



2GHz CPU

clock period = $\frac{1}{f} = \frac{1}{2\text{GHz}}$

= .5ns

SSD → $10^6$
HDD → $10^7$


HEB
fridge
home in the mountains....
20 miles
need butter
go to HEB
buy 8 tubs
high rise apartment
need butter!
go to convenience store
buy just one tub
convenience store

# Mmap()

- Library:

  include <sys/mman.h>
- Syntax:

  void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off);
- Usage pa = mmap(addr, len, prot, flags, fildes, off);

Establishes a memory map of file given by fildes (file descriptor), starting from offset off, of length len, at memory address starting from pa

file descriptor

effectively "map" the desired portion of the file on disk (slow) to memory (faster)

# Mmap()

- prot controls read/write/execute accesses of memory

PROT_READ: read permission

PROT_WRITE: write permission

PROT_EXECUTE: execute permission

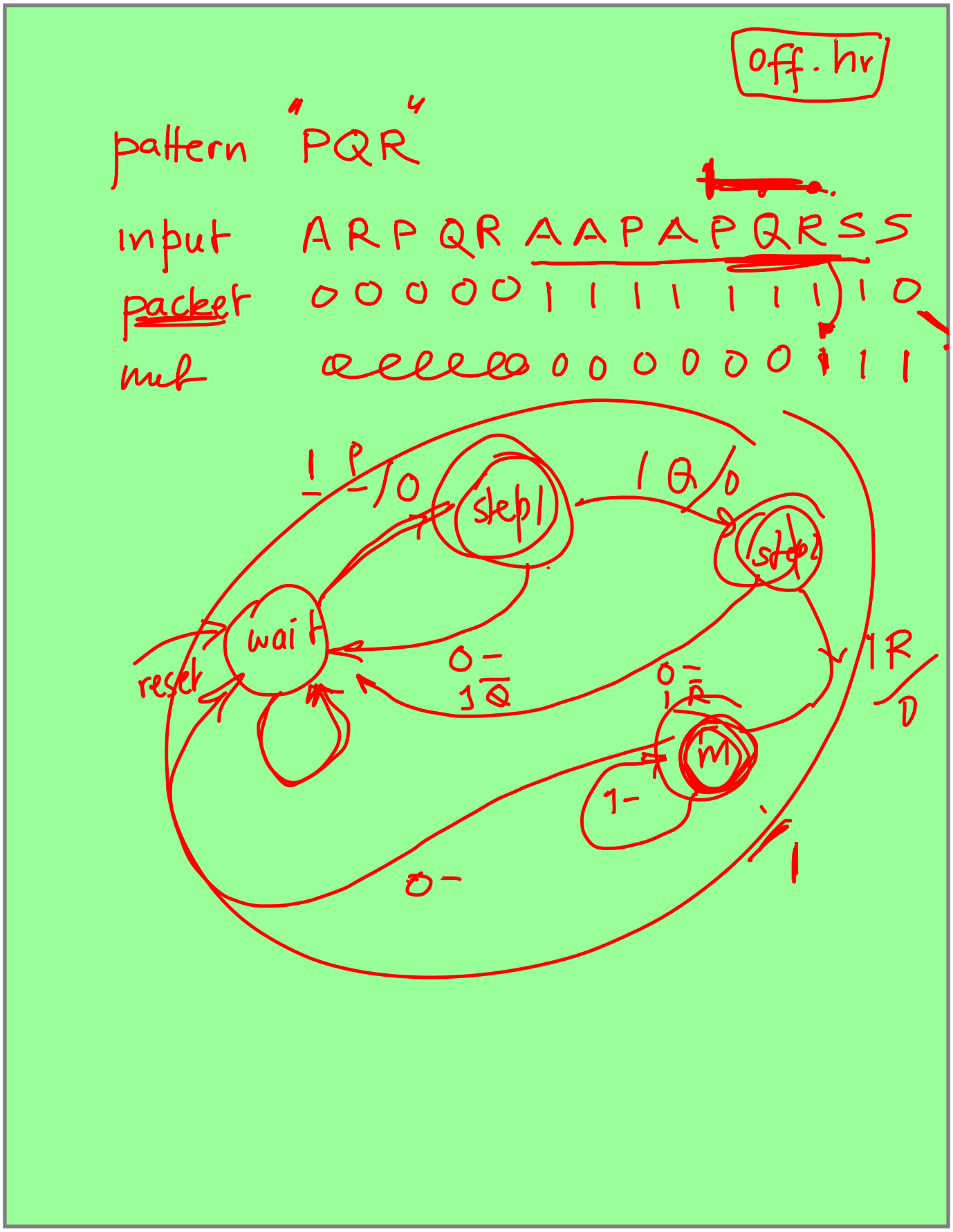
PROT_NONE: no permission

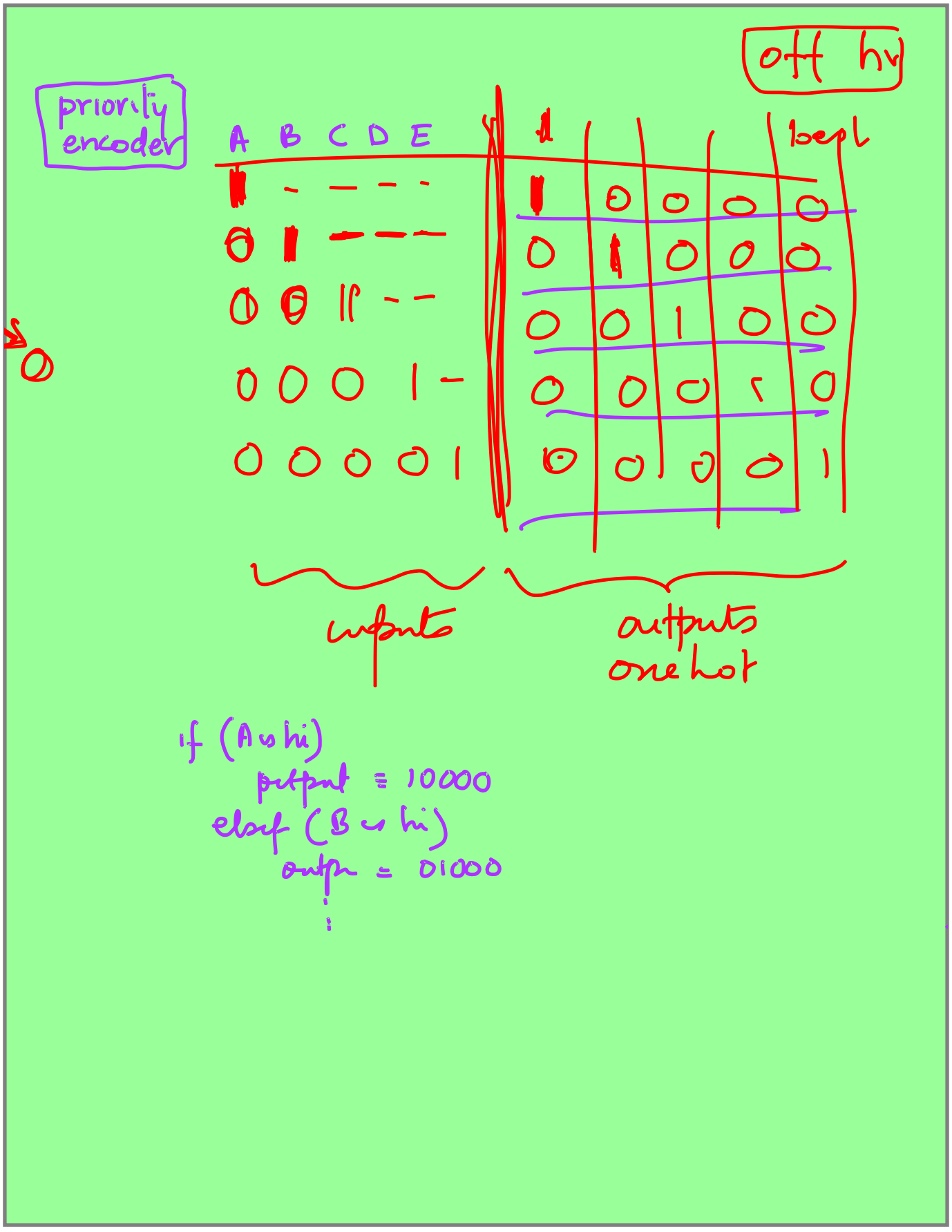
Can use OR combination of these bits

- Flags filed controls the type of memory handling

MAP_SHARED: changes are shared

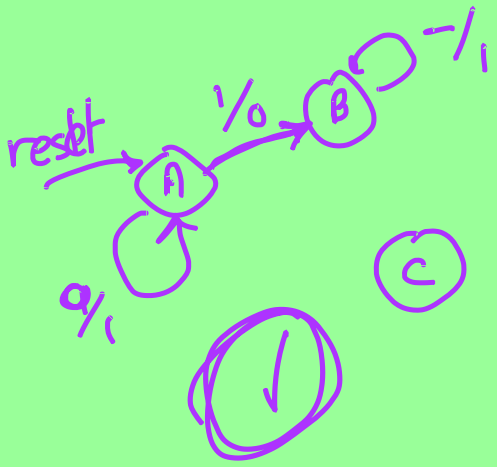
MAP_PRIVATE: changes are not shared

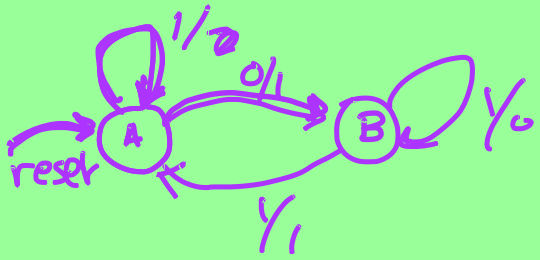

off.hr
pattern "PQR"
input A R P Q R A A P A P Q R S S
packet 0 0 0 0 0 1 1 1 1 1 1 1 1 0
mc
1 P
1 Q
step1
step2
wait
reset
0 -
1 Q
0 -
1 R
1 R
D
1 -
0 -
1

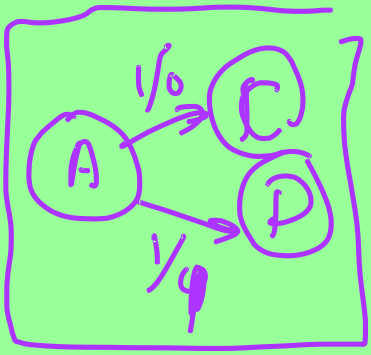

priority encoder
off by
A B C D E
1 - - - -
0 1 - - -
0 0 1 - -
0 0 0 1 -
0 0 0 0 1
1 0 0 0 0
0 1 0 0 0
0 0 1 0 0
0 0 0 1 0
0 0 0 0 1
inputs
outputs
one hot
if (A is hi)
output = 10000
elsif (B is hi)
output = 01000

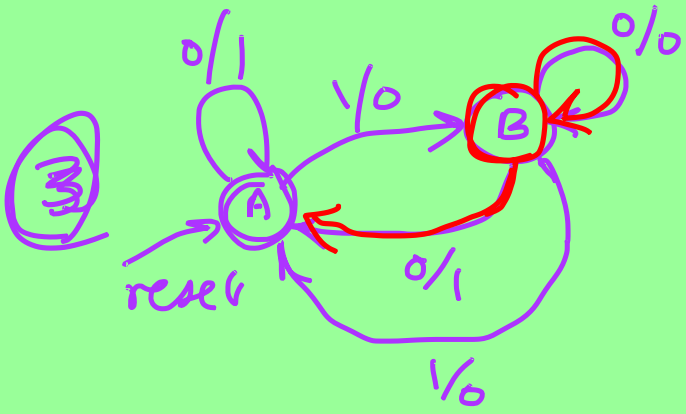
off. hr

Q: what is an unreachable state




| | | 1 | 2 | 3 |
|---|---|---|---|---|
| Q1 | are all states reachable | NO (∵ c) | Yes | Yes |
| Q2 | are the FSMs deterministic | Y | Y | N |