Functions and I/O

For use in CSE6010 only

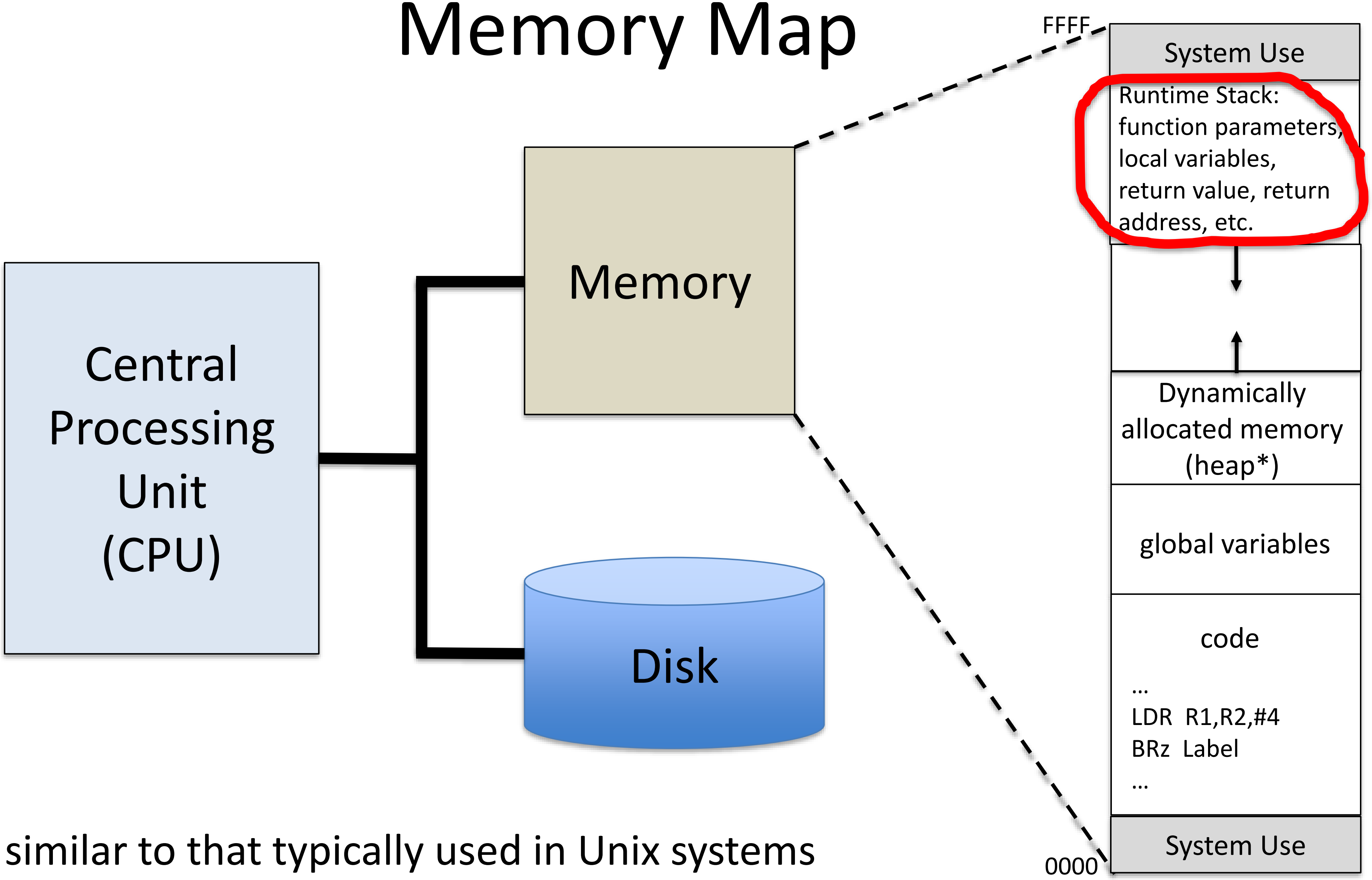
Not for distribution

Levels of Abstraction in Computers

Application (word processor, simulator, web browser, …)	☑
Algorithms (sorting, optimization, equation solver, …)	☑
Programming Language (C, FORTRAN, Matlab, Java, …)	☑
Operating System (UNIX, Windows, iOS, …)	
Machine Instruction Set Architecture (Intel i86, ARM, …)	☑
Machine Organization (Main memory, registers, adders, …)	☑
Logic Gates (NAND, NOR, inverter, …)	☑
Transistors (CMOS, NMOS, …)	☑
Physics (Semiconductors) (electrons, holes, …)	☑

Programming language implementation

Memory Map



- Layout similar to that typically used in Unix systems
- Runtime stack and heap grow/shrink during program execution

* Not related to heap data structure

Structure of a Typical C Program

```
int a, b, c;                /* global variables */
// function prototypes go here

main ()
{   int i, j, k;
    i=1; j=2; k=3;
. . . foo(i,&j) . . .          /* function call (invocation) */
}

int foo (int x, int *y)  /* function definition */
{   int a;                         /* local variable */
    a=4;                           /* reference local variable */
. . . bar(a); . . .
    return (a);
}

void bar (int w)
{   double i;
. . . reference a . . .        /* reference to global*/
}
```

arguments: actual values passed

parameters: variables in func definition

Parameter Passing: Call By Value

Call by Value (used in C)

- A *copy* of the argument is created, passed to function
- Upon return, the copy is discarded
- Implications of copying
  - Copying can be expensive (time and space) for large parameters
  - Modifications to the parameter within the function are made to the copy, and are not visible to the calling program
- Q: What if you want modifications to be visible to the caller?
- A: Pass a *pointer* to the data being passed; value passed is an address, e.g., `scanf ("%d", &i);`
- Aside: in C, array arguments pass a pointer to the array

Other approaches: call by value return, call by reference, call by name

Call by Value Example

```
#include <stdio.h>
int foo (int x, int *y);   // integer, pointer to integer

main()
{   int i, j, k;

    i=1; j=2;  k=3;
    k = foo (i, &j);
    printf ("i=%d, j=%d\n", i, j);
}

int foo (int x, int *y)
{
    x=3;
    *y = 4;
    return (…)
}
```

What is printed?

i=1, j=4

Calling a Function

- What is needed to properly implement a function call and the return to the main program?

Calling a Function

- What is needed to properly implement a function call and the return to the main program?
  - Keeping track of where we are in the program: need to know the address where the function begins, the address of the next instruction in the main program (incremented PC) for returning, and the values of local variables in the main program
  - Variable space for the function: track arguments passed, variables returned; know and allocate space for local variables used in the function
- We will be talking about both of these.

Implementing Function Call / Return

```
main()
{
    k = foo (i, &j);
}

int foo (int x, int *y)
{
    return (…)
}
```

- Function Call
  - Similar to branch instruction
  - Execution "jumps" to the code implementing the function
- Function Return
  - Where does execution continue after the function is completed?
- Implementation solution
  - Function call saves the return address (where to resume)
  - Jump to the saved return address to return from the function

LC-3 Machine Instructions

LC-3: save the return address in register R7

JSR offset // Jump Subroutine (=function) instruction
// R7 <- PC+1 (return address)
// PC <- PC+1 + offset (sign extended)

opcode

0	1	0	0	1	offset (11 bits)
15					0

RET // PC <- R7
// special case of the JMP instruction (R7)

opcode

1	1	0	0	0	0	0	1	1	1	0	0	0	0	0	0
15															0

Function Call and Return

```
main ()                JSR foo
{
. . . foo(i,&j) . . .
}

int foo (int x, int *y)
{
  . . . bar(a); . . .
}

void bar (int w)
{
. . .
}
```

JSR foo // Save return address (PC+1) in R7

Where do we save the return address for bar?

RET // Resume execution of caller (jump to address in R7)

The Runtime Stack

- We can consider memory allocation for the return address for function calls like list data structure inserts and deletes
  - Function call: insert return address into a list
  - Function return: delete return address from list
- When a function returns, the most recently saved return address should be used
- This suggest a last-in-first-out approach, i.e., a stack

Storage for function calls is managed in an area of memory called the runtime stack.

LC-3 Function Call / Return

```
main ()
{
. . . foo(i,&j) . . .
}

int foo (int x, int *y)
{
. . .
bar(a);
. . .
}

void bar (int w)
{
. . .
}
```

JSR foo (save return address in R7)

Resume execution here

PUSH R7

JSR bar (save return address in R7)

Resume execution here

R7<-POP

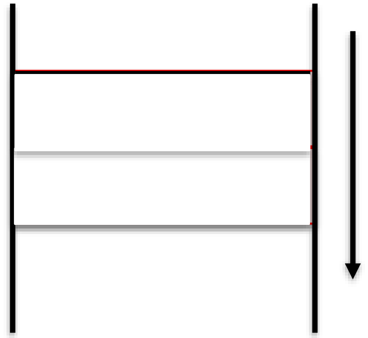
RET (PC<-R7)

PUSH R7

R7<-POP

RET (PC<-R7)

Runtime Stack



LC-3: Register R6 used as pointer to top of stack

We're only just getting started!
Functions need additional storage.

Other storage must be allocated with each function call, and deallocated when the function returns

- Parameters/arguments passed to the function
  - Caller copies argument values into this storage
  - The callee (function being called) needs to access parameters
- Local variables declared within the called function
  - Allocated with each function call, released upon return
  - A function can call itself (recursion)
- Value returned by the function
  - Callee writes return value into this storage
  - Caller must be able to access the return value

All of these items are also stored on the runtime stack

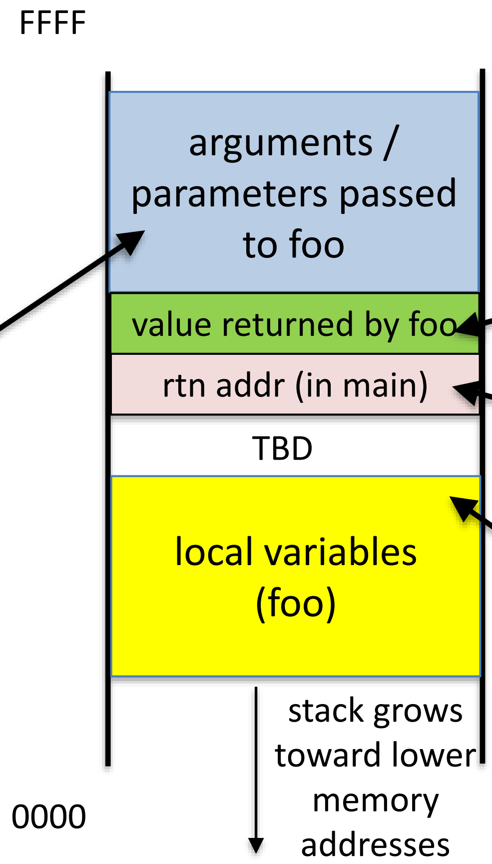
Stack Frame (Activation Record)

Caller Function

```
main ()
{
. . . foo(i,&j) . . .
}
```

1. Caller pushes arguments onto stack

2. Caller executes “JSR foo”



Callee Function

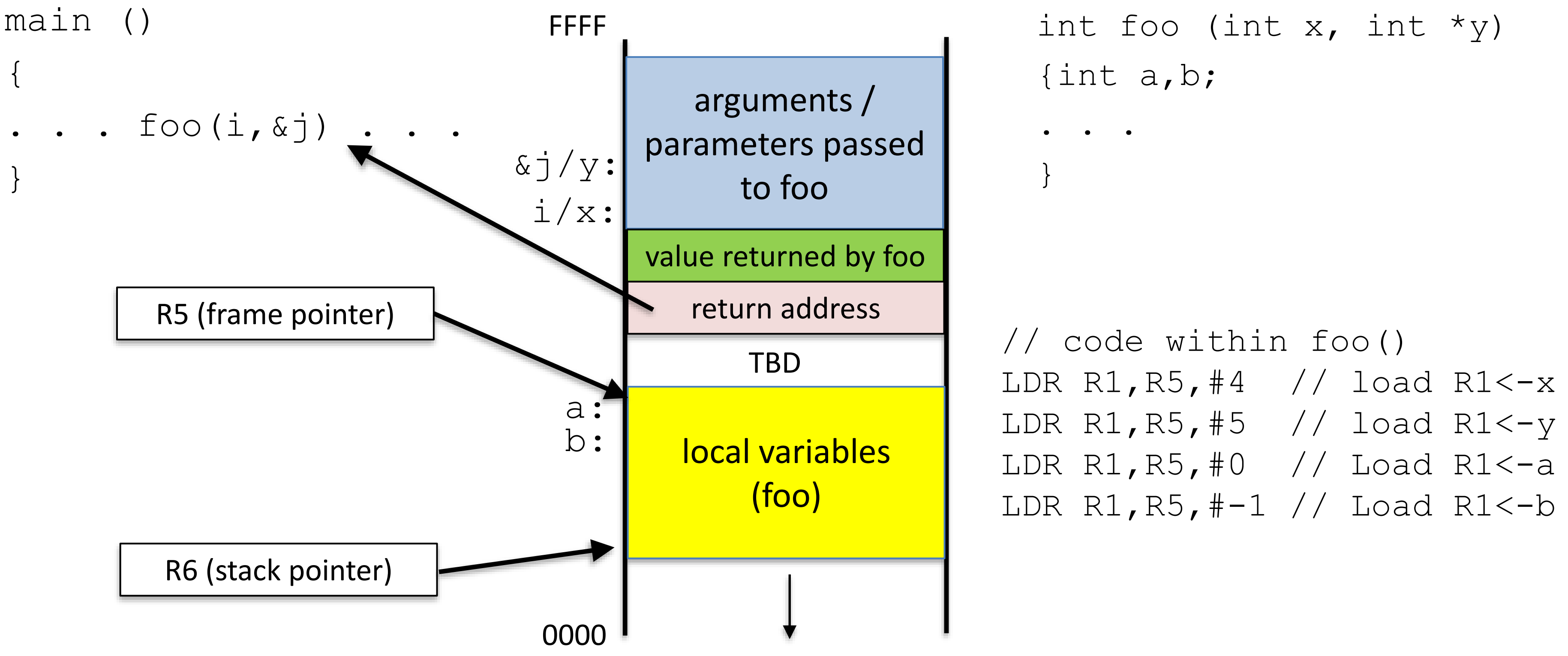
```
int foo (int x, int *y)
{int a;
. . .
}
```

3. Callee (foo) allocates space for value returned by foo
4. Callee (foo) pushes return address (in R7)

5. Callee (foo) allocates space for local variables

- Holds parameters/arguments, local variables, return value, return address for an invocation (call) of a function
- One frame associated with each execution (invocation) of a function
  - At any instant, there is a single “current” frame being used
  - New frame pushed onto runtime stack with each function call
  - Pop frame from stack with each function return

Frame Pointer Register



- Each function invocation needs a way to access its parameters, local variables, return value (i.e., its stack frame)
- A CPU register (frame pointer register) holds a pointer to the current stack frame
  - LC-3 uses register R5
  - R5 points to first local variable in stack frame
  - Use indexed addressing mode to access locals, parameters, return value (sound familiar?)

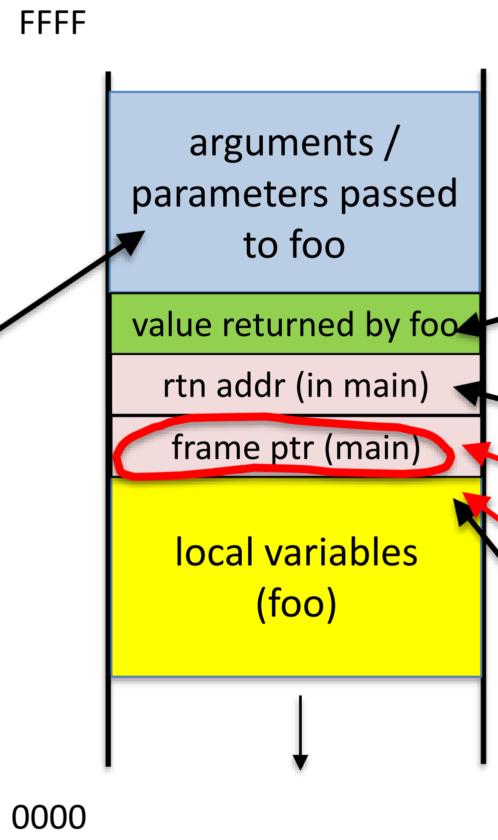
Stack Frame (Revised)

Caller Function

```
main ()
{
. . . foo(i,&j) . . .
}
```

Callee Function

```
int foo (int x, int *y)
{int a,b;
. . .
}
```



1. Caller pushes arguments onto stack

2. Caller executes “JSR foo”

3. Callee (foo) allocates space for value returned by foo

4. Callee (foo) pushes return address (in R7)

4a. Callee (foo) pushes R5, the frame pointer for caller (main)

4b. Callee (foo) resets R5 to point to the new (foo’s) frame

5. Callee (foo) allocates space for local variables

- The frame pointer points to the stack frame for the function now being executed
- Function call: need to save the frame pointer for the caller on the stack (in addition to the return address)

Function Call Example

```
main()
{       int i, j, k;
…
        i=1; j=2; k=3;
        k = foo (i, &j);
…
}

int foo (int x, int *y)
{
        int a;
…
        x=3; *y=4;
        a=4;
        bar (a);
…
        return (a);
}

void bar (int w)
{
…
}
```

Call (invoke) function foo():

- Push &j onto stack
- Push i onto stack
- `JSR foo` (save PC+1 in R7)

Execute function `foo`():

- Allocate storage on stack to hold the value returned by the function
- Push R7 (return address)
- Push R5 (frame pointer for `main`)
- Set R5 to point to new stack frame
- Allocate storage for local variables (a)
- Execute body of function (access local variables, parameters, etc.)

Return:

- Copy return value into return value storage
- Pop (release storage) for local variables
- Pop (restore) frame pointer into R5
- Pop return address into R7
- `RET` (jump to address in R7)

Resume execution after call:

- Pop (release storage) return value
- Pop (release storage) for parameters

Runtime Stack Example

```
main()
{   int i, j, k;
…
    i=1; j=2; k=3;
    k = foo (i, &j);
…
}
```

After foo call, i is still 1;
j has been changed to 4

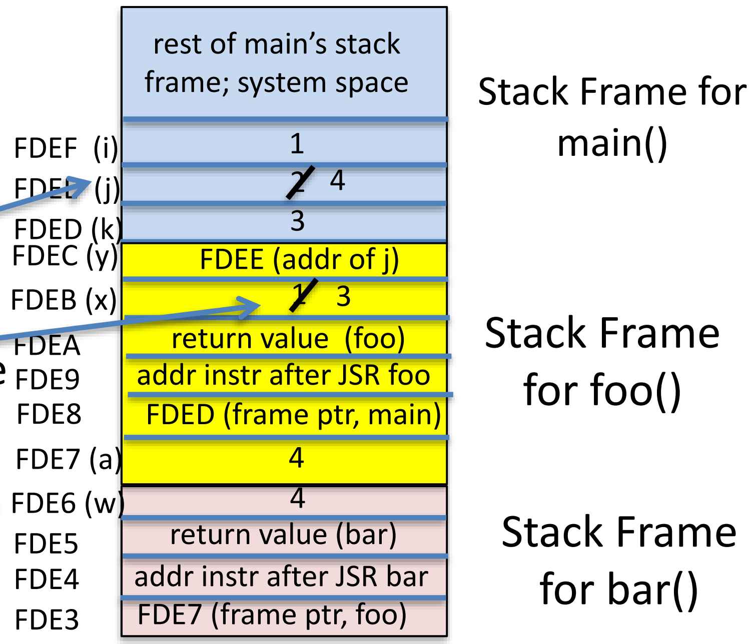
```
int foo (int x, int *y)
{
    int a; …
    a=4;
    x=3;
    *y=4;
    bar (a);
…
}

void bar (int w)
{
…
}
```

Push new stack frame when foo is called

Pop stack frame when foo returns

Push new stack frame when bar is called

Pop stack frame when bar returns



Note that addresses *within an activation record* and *on the runtime stack* usually decrease

- Function call: push new frame onto stack; this becomes the current frame
- Return: Pop current frame from stack

```
main()
{ int i; // one local variable
```

1. Pre-function call

- Push arguments onto stack
- `JSR foo` (save PC+1 in R7)

```
  foo (i);
```

4. Post-function return

- Pop (release storage) return value
- Pop (release storage) for arguments

```
}
int foo (int i)
{ int a;
```

2. Post-call, Pre-function execution

- Allocate storage for return value
- Push R7 (return address)
- Push R5 (frame pointer for `main`)
- Set R5 to point to new stack frame
- Allocate storage for local variables
- Execute body of function

3. Post-execution, Pre-function return

- Copy return value into return value storage
- Pop (release storage) for local variables
- Pop (restore) frame pointer into R5
- Pop return address into R7
- `RET` (jump to address in R7)

```
}
```

```
// R6: stack pointer (SP)
// R5: frame pointer (FP)
LDR R0,R5,#0    // R0<-i
ADD R6,R6,#-1   // Push i
STR R0,R6,#0    // M[R6]<-R0
JSR foo
LDR R0,R6,#0    // R0<- rtn val
ADD R6,R6,#1    //
ADD R6,R6,#1    // pop arguments

foo: ADD R6,R6,#-1   // return value
ADD R6,R6,#-1   // Push R7
STR R7,R6,#0    // SR, BR, off
ADD R6,R6,#-1   // Push R5 (FP)
STR R5,R6,#0
ADD R5,R6,#-1   // Set FP
ADD R6,R6,#-1   // One local var
// assume return value in R0
STR R0,R5,#3    // return value
ADD R6,R5,#1    // pop local var
LDR R5,R6,#0    // pop FP
ADD R6,R6,#1
LDR R7,R6,#0    // pop ret addr
ADD R6,R6,#1
RET             // return
```

Loads function argument

Decreasing SP by 1 adds space for 1 local variable; pushes the function argument i onto stack

Load return val (top of stack)

Pop return value (SP++)

Pop arguments (SP++)

Decrementing SP to make space for return val (SP--)

Store return address (SP--)

Push caller's FP (SP--)

New FP

Memory for foo's local var (SP--)

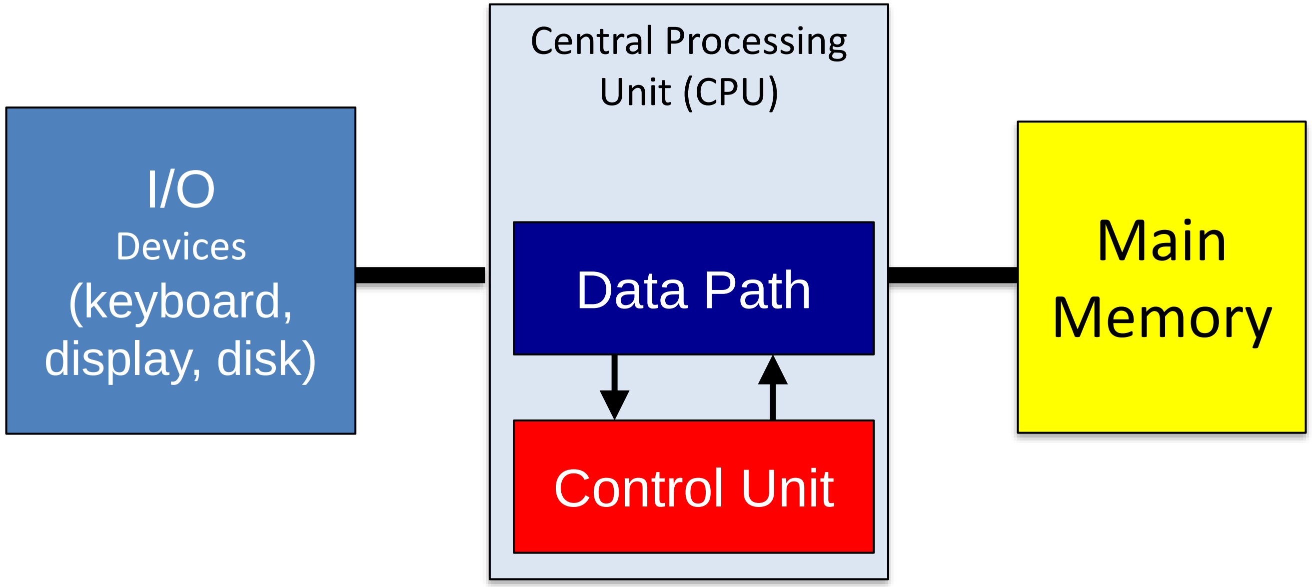
Store return val

Pop items on stack: local var, FP, ret addr (SP++)

Summary: Functions

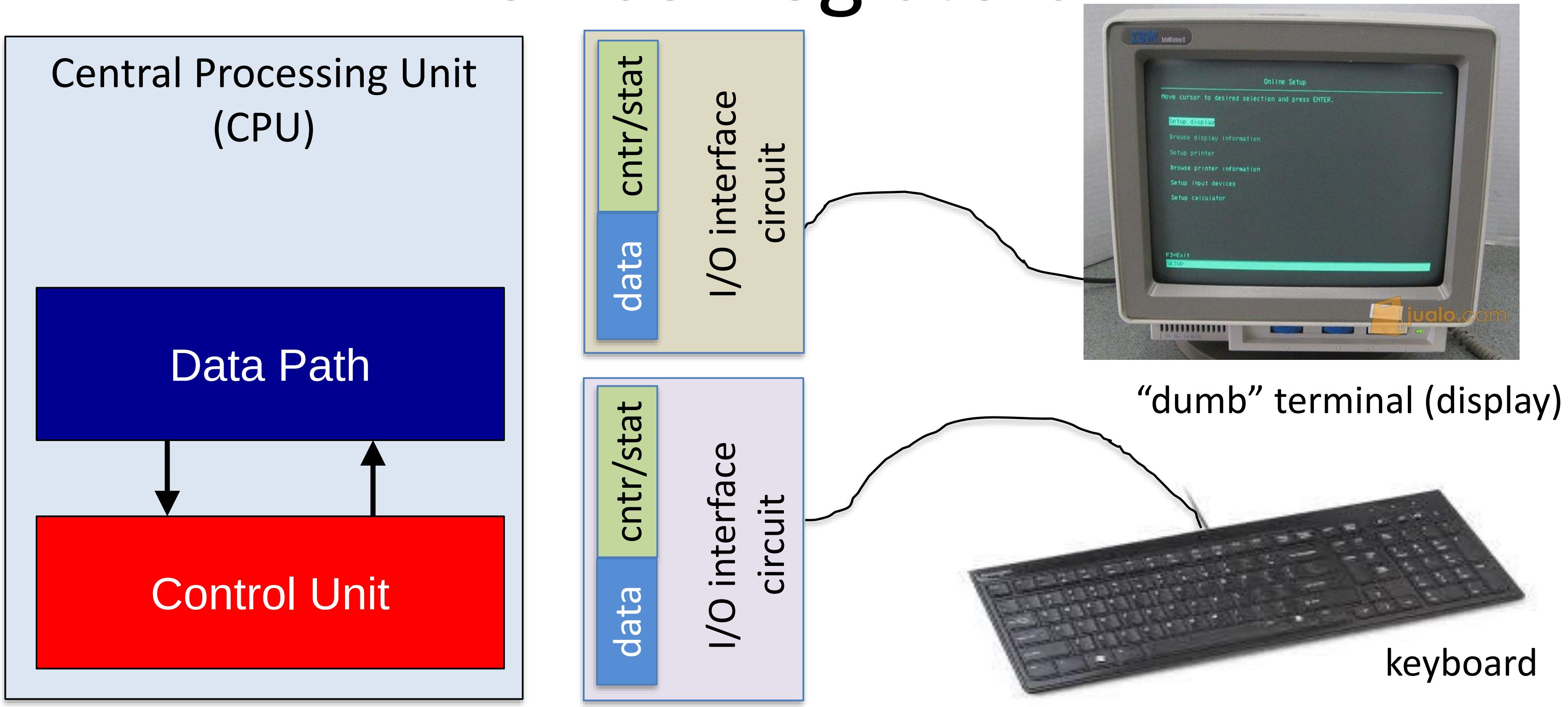
- C uses call by value; be careful whether to pass a variable or a pointer
  - If argument is to be modified by function, pass a pointer
  - If the argument is large (e.g., a large structure), pass a pointer
  - Otherwise, use the variable itself as the argument
- C (and many other language) implementations use a runtime stack to hold parameters, local variables, and other information
  - Stack needed to implement recursion
- Function call mechanism will impact performance
  - A little time needed for function call/return (overhead)
  - Memory also required for stack; implications on recursion
  - In a multi-threaded program, each thread has its own runtime stack, introducing some complexity

Von Neumann Machine Model



- Most Input/Output (I/O) devices are EXTREMELY slow compared to CPU
  - Disk: read/write ~1,000,000 times slower than CPU register
  - Keyboard/display: much slower than disk!
  - Doesn't make sense for CPU to remain idle waiting for I/O operations to complete
- I/O devices operate concurrent with CPU
  - How does the CPU know when an I/O operation has completed?

Device Registers



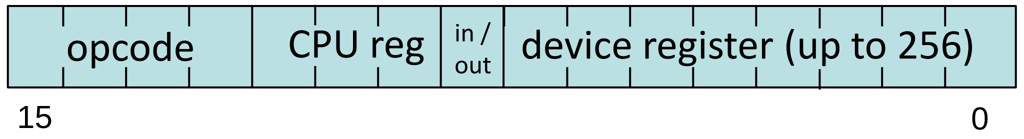
CPU communicates with I/O devices through device registers

- Data (transferred to/from device)
- Control: CPU writes into this register to (for example) initiate an I/O operation and specify parameters (e.g., what disk locations)
- Status: CPU reads this register to find out the status of I/O operations (has data been received? I/O errors?)

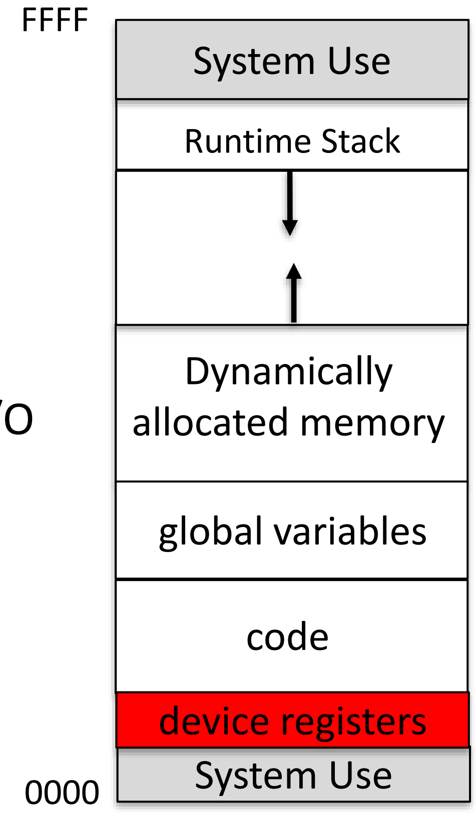
Machine Instructions to Access Device Register

- I/O instructions
  - Machine instructions defined specifically to read/write device registers
  - Separate "address space" for I/O devices
- Memory-Mapped I/O (more common)
  - Each device register assigned to a specific memory address
  - Existing instructions (e.g., LDR, STR) used to read/write device registers

Example CPU I/O instruction

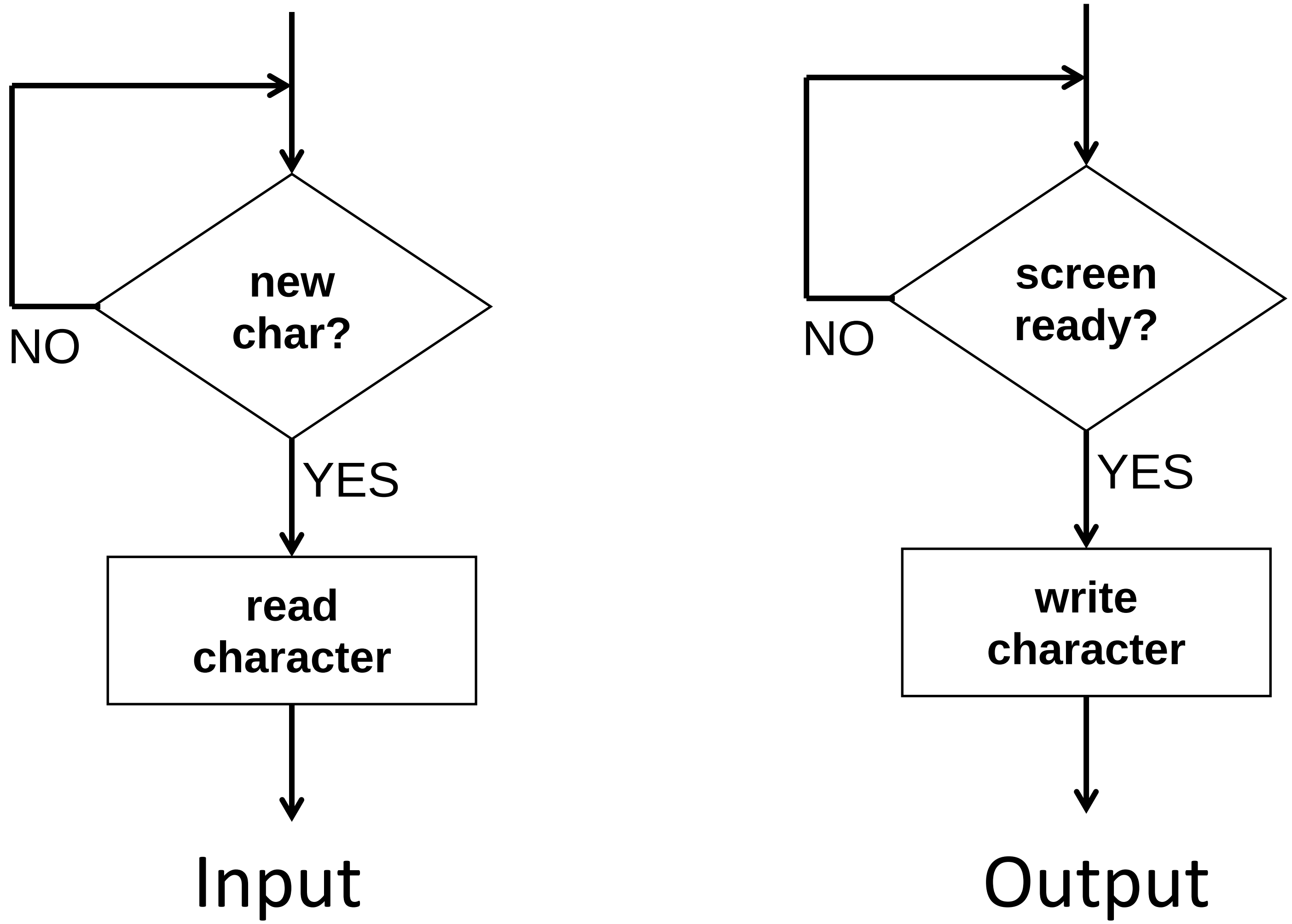


Example:
Memory mapped I/O

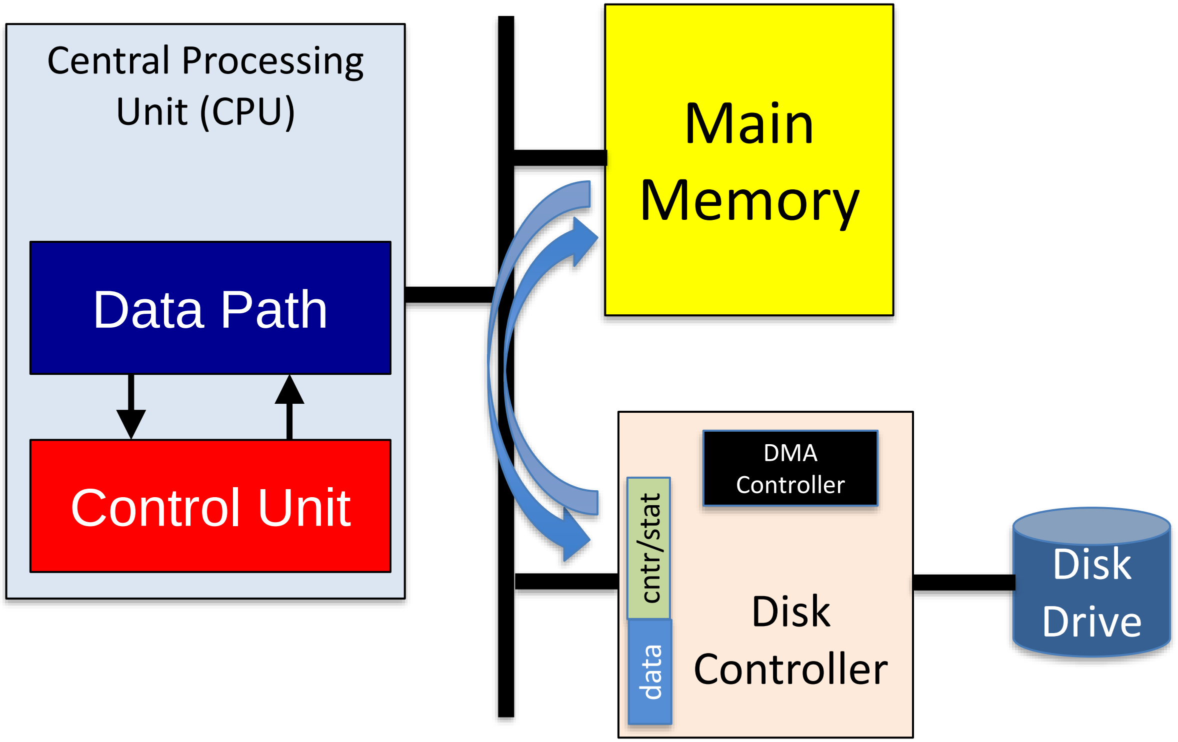


Programmed I/O: Polling

Low level I/O software usually implemented as functions within the operating system (device drivers)



Direct Memory Access (DMA)



- High-speed devices: programmed I/O uses too much CPU time
  - Disk drives (rotating, solid state): read/write ~4K bytes at a time
  - High-speed network interface
  - Graphics (may use main memory or memory within graphics card)
- Programmed I/O used to start/finish I/O operations
- Data: Device directly accesses main memory independent of the CPU
- Leads to an alternative to polling – next topic!

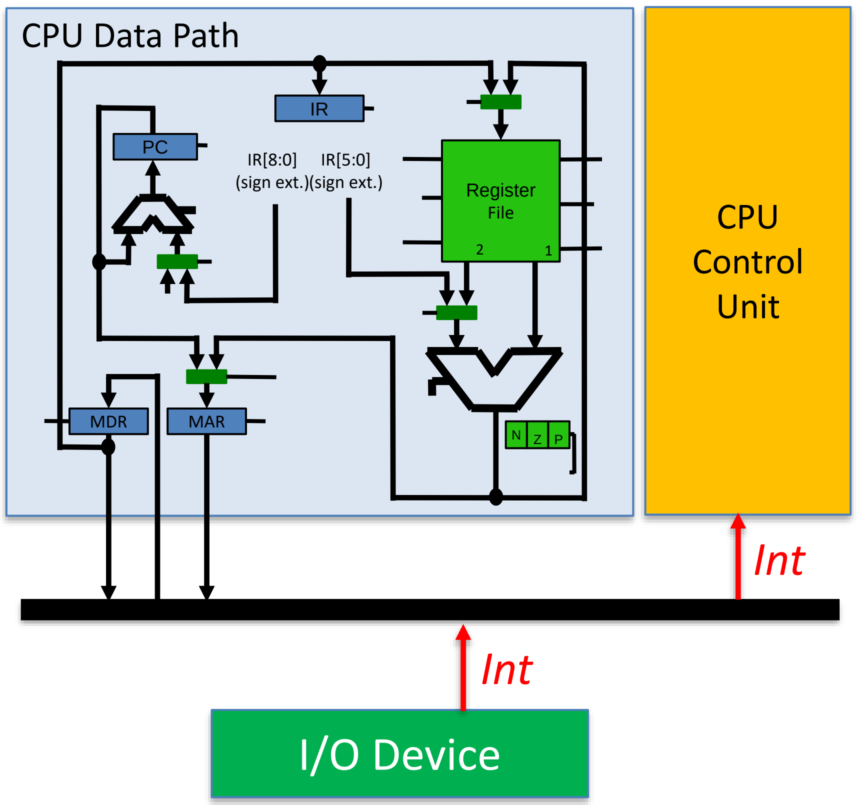
Polling vs. Interrupts

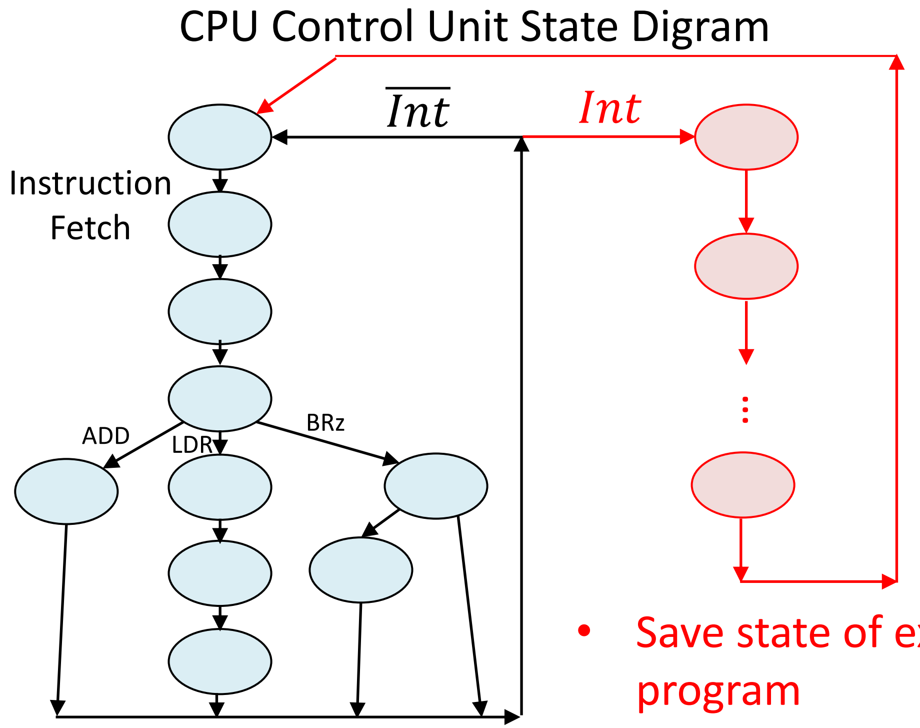
- Polling
  - CPU keeps checking status register until *new data* arrives OR *device ready* for next data
  - "Are we there yet? Are we there yet? Are we there yet?"
- Interrupts
  - Device sends a special signal to CPU when *new data* arrives OR *device ready* for next data
  - CPU can be performing other tasks instead of polling device.
  - "Wake me when we get there."

- To implement an interrupt mechanism, we need:
  - A way for the I/O device to signal the CPU that an interesting event has occurred
  - A way for the CPU to test whether the interrupt signal is set
  - A way to identify which device generated the interrupt

Implementing Interrupts: Hardware

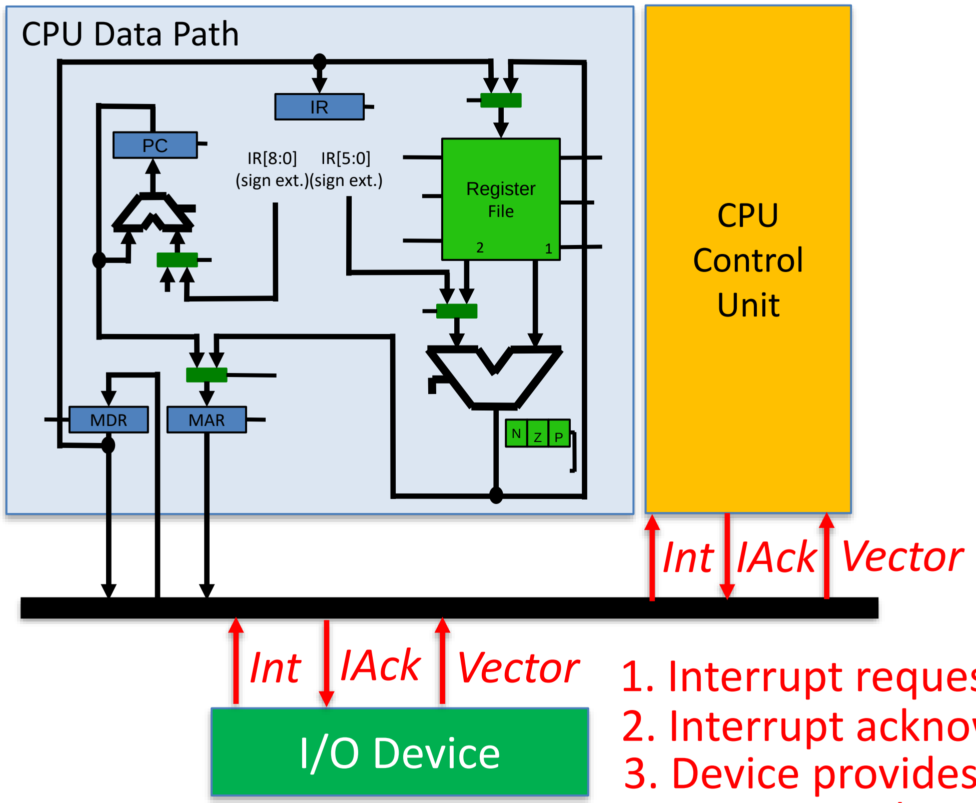
- Add Interrupt Status Signal (*Int*)
  - Asserted by I/O device to trigger an interrupt
- Hardware checks *Int* before each instruction fetch
- If *Int* is asserted
  - Execute a sequence of steps to process interrupt
  - Resume normal instruction execution

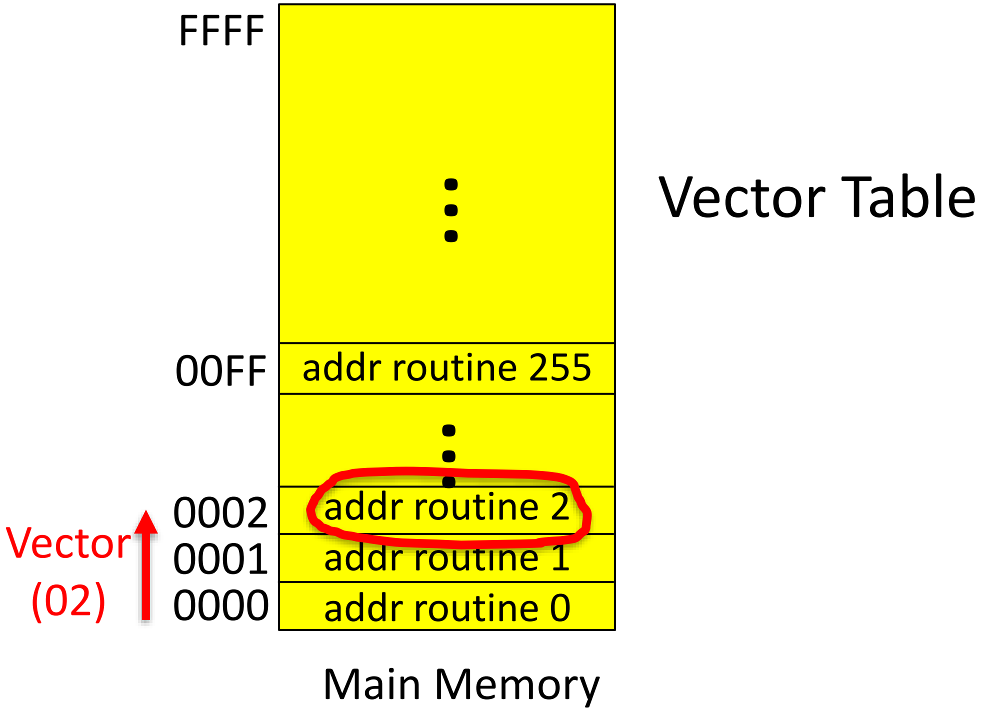




- Save state of executing program
- Jump to interrupt routine

Vectored Interrupts





1. Interrupt request
2. Interrupt acknowledgement
3. Device provides Vector (e.g., 02)
4. Save PC and CC registers (where?)
5. PC <- M[vector]

On a (special) stack!

- Device provides an interrupt vector to identify itself (e.g., 8 bits)
  - Each device assigned a unique vector
- Vector used as index into table that holds the *starting addresses* of interrupt service routines
  - Location of table often fixed (defined by hardware) [e.g., 0000 to 00FF]

Interrupts: Saving Processor State

- The program being interrupted does not know it is being interrupted!
- Interrupt mechanism must save enough state from the currently executing program so its execution can be resumed later
  - LC-3: PC and CC
  - Other state (e.g., registers R0-R7 saved by the interrupt handler software if the interrupt handler uses them)
- Return to the interrupted program
  - Restore state of the interrupted program (PC, CC)
    - PC <- address of the interrupted instruction + 1
  - Return often implemented with a special machine instruction
    - RTI: Return from Interrupt (LC-3; restores CC register)

Other Aspects

- An interrupt handler might itself be interrupted by another device (nested interrupts)
  - Use “system” runtime stack (separate from the “user” runtime stack discussed earlier)
- An interrupt handler might not want to be interrupted
  - CPUs have the ability to “turn off” interrupts
  - Interrupt priorities often used to determine which interrupts can interrupt which interrupt handlers

When to Use Interrupts

- When timing of external events is uncertain
  - Example: incoming packet from network
- When device operation takes a long time
  - Example: start a disk transfer, disk interrupts when transfer is finished
  - processor can do something else in the meantime
- When event is rare but critical
  - Example: machine is losing power!

Traps

- A Trap is essentially the same as an interrupt, except it is *generated internally by the CPU* rather than by an external device
- User program requests the OS to perform a task on behalf of the program; OS takes control, handles the request, and returns control to the user program
- Examples
  - System call, e.g., to initiate I/O operations
    - LC-3: TRAP instruction defined for this purpose
    - Like a function call, except the processor changes to "system mode" which can access device registers, and perform other functions user code cannot
    - Execution returns to user program after system call completed
  - Program execution errors
    - Divide by zero, invalid memory reference, etc.
    - Execution jumps to operating system code to "kill" program

Summary: I/O

- Key concepts (vocabulary)
  - Device registers
  - Memory-mapped I/O
  - Polling
  - Direct memory access (DMA)
  - Interrupts
  - Interrupt vector
  - Trap
- Your program routinely gets interrupted at unpredictable times, affecting the timing of its execution
  - Race conditions in parallel programs
- I/O, interrupt handlers, traps, programmed by system software developers