Lecture 18

COP3402 FALL 2015 – DR. MATTHEW GERBER – 11/23/2015

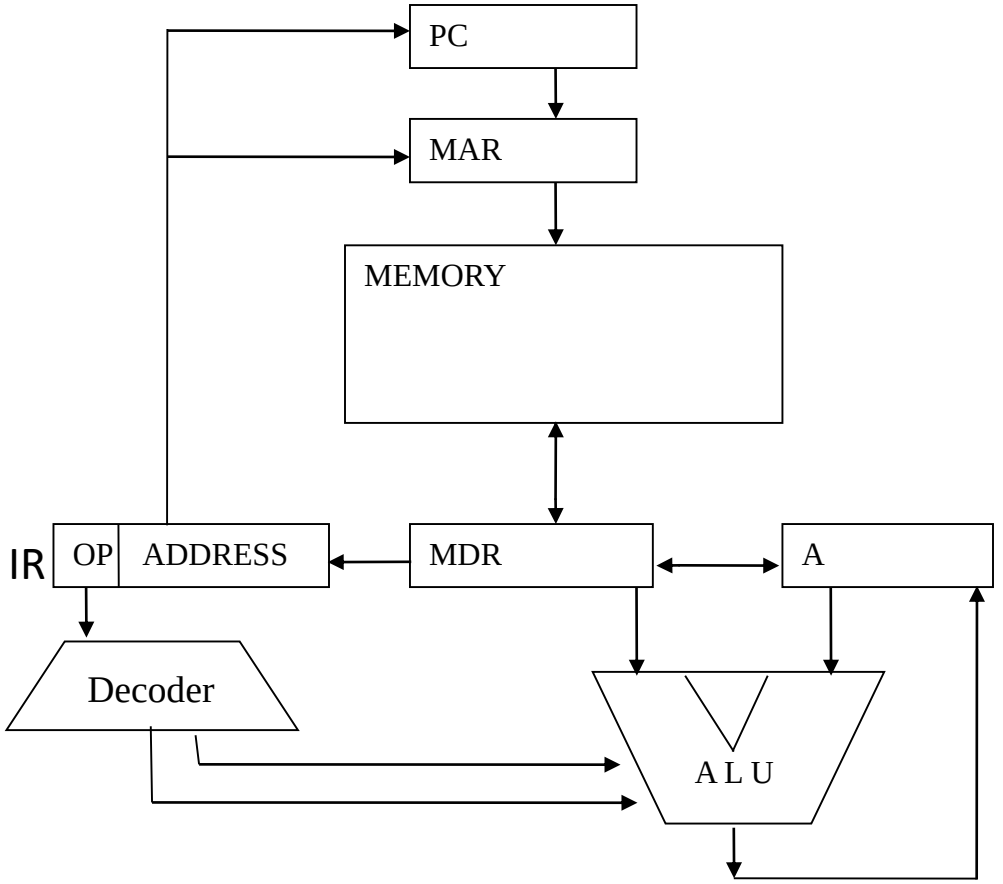
FROM EURIPIDES MONTAGNE, FALL 2014

Tonight

- Review: The Tiny Computer
- The Interrupt Cycle
- Memory Protection
- Privileged Instructions
- System Calls

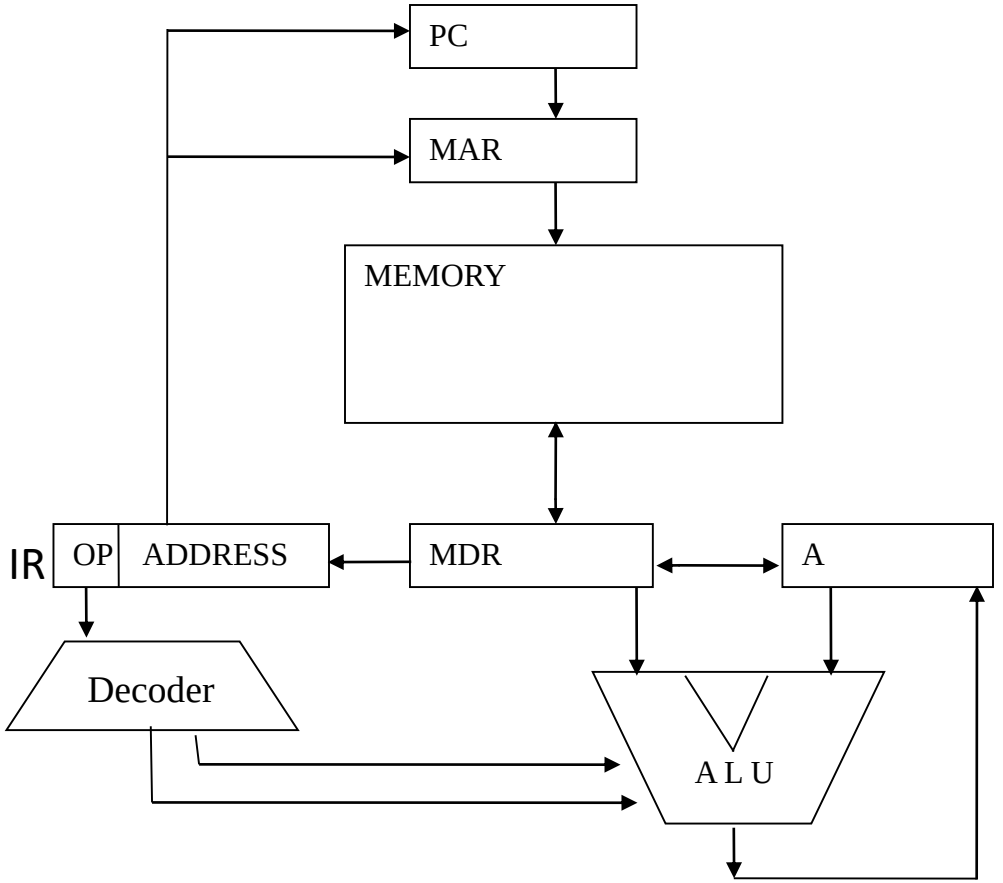
Definitions, Part 1: Components

- **Main Storage (MEMORY):** Used to store programs and data; in modern systems this is invariably implemented as RAM
- **Arithmetic Logic Unit (ALU):** Performs the actual execution of instructions
- **Decoder:** Examines the IR (next slide), decides which instruction the processor will execute and signals the ALU to execute it



Definitions, Part 2: Registers

- **Program Counter (PC):** Holds the address of the next instruction to be executed
- **Memory Address Register (MAR):** Used to specify the address to a specific memory location in main storage
- **Memory Data Register (MDR):** Used to store data being sent to or received from main storage
- **Instruction Register (IR):** Stores the instruction to be executed by the processor
  - Has two components that we'll discuss later
- **Accumulator (A):** Stores data to be used as input to the ALU



The Instruction Cycle

The *instruction cycle*, *machine cycle* or *fetch-execute cycle* of the Von Neumann machine has two looping steps:

- **Fetch Cycle:** Retrieve an instruction from memory
- **Execute Cycle:** Execute the retrieved instruction

When the instruction is finished executing, the next instruction is fetched and the cycle continues.

The Tiny ISA

0 Fetch (hidden instruction)

MAR ←PC

MDR ←MEM[MAR]

IR ←MDR

PC ←PC+1

DECODER ←IR.OP

2 ADD

MAR←IR.Address

MDR ←MEM[MAR]

A ← A + MDR

DECODER ←00

1 LOAD

MAR←IR.Address

MDR ←MEM[MAR]

A ← MDR

DECODER←00

3 STORE

MAR←IR.Address

MDR ←A

MEM[MAR] ←MDR

DECODER ←00

7 HALT

Op	Mnemonic	Meaning
0	(Fetch cycle)	Fetch next instruction
1	LOAD <addr>	Load <addr> contents into A
2	ADD <addr>	Add <addr> contents into A
3	STORE <addr>	Store A contents into <addr>
4	SUB <addr>	Sub <addr> contents from A
5	IN <device>	Read into A from <device>
6	OUT <device>	Write to <device> from A
7	HALT	End program
8	JMP <addr>	Branch to <addr>
9	SKIPZ	Skip next instruction if Z (A = 0)
A	SKIPG	Skip next instruction if G (A > 0)
B	SKIPL	Skip next instruction if L (A < 0)

Error Detection

So far, we have a computer that can execute programs – if they're error-free.

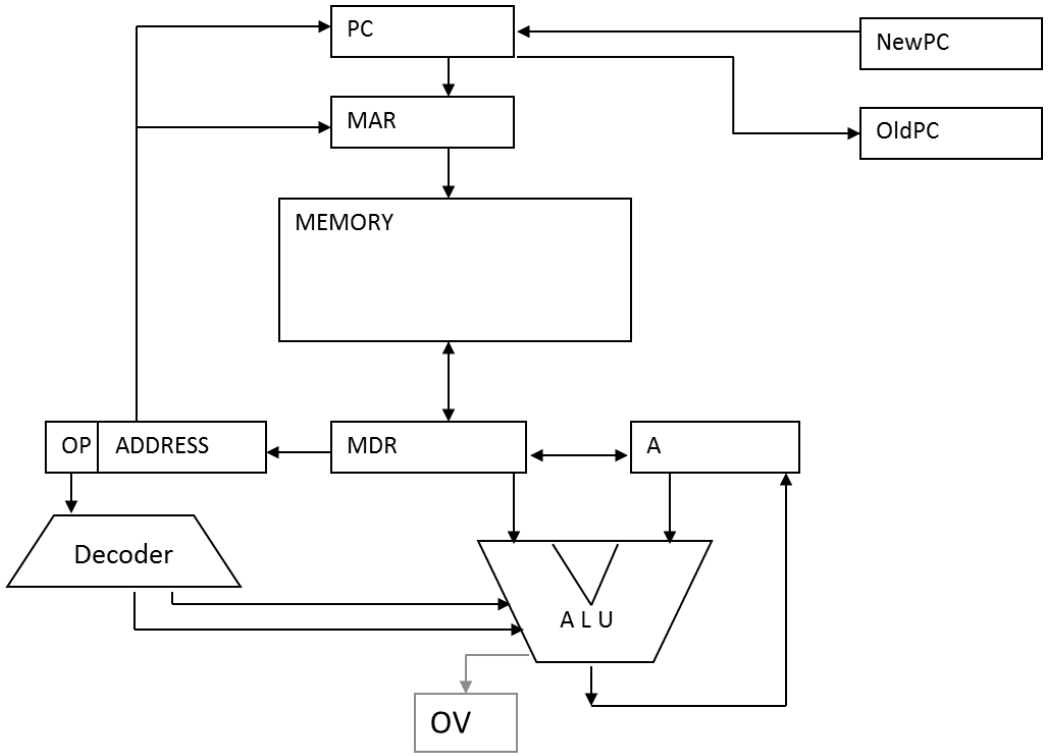
- What would happen if an overflow occurred while executing an addition operation?
- Let's think about a way to detect this type of event and act accordingly…

We can easily add a hardware flip/flop to the ALU that detects overflow.

- So far, so good
- But what do we *do* about it?

We need to detect this condition, and when it occurs, end the program abnormally.

- The Fetch/Execute cycle becomes a *Fetch/Execute/Interrupt* cycle.



The Interrupt Cycle

0 Fetch (hidden instruction)

MAR ←PC

MDR ←MEM[MAR]

IR ←MDR

PC ←PC+1

DECODER ←IR.OP

2 ADD

MAR←IR.Address

MDR ←MEM[MAR]

A ← A + MDR

DECODER ←0F

1 LOAD

MAR←IR.Address

MDR ←MEM[MAR]

A ← MDR

DECODER←0F

3 STORE

MAR←IR.Address

MDR ←A

MEM[MAR] ←MDR

DECODER ←0F

7 HALT

Add a second hidden instruction (opcode F).

- Now, at the end of each execution cycle, the decoder is changed to F instead of 0

Instruction F is pretty simple:

F Abend (hidden instruction)

If OV=1

then HALT

DECODER ← 00

Interrupt Handling

Instead of simply halting execution we can transfer to an *interrupt handler routine*.

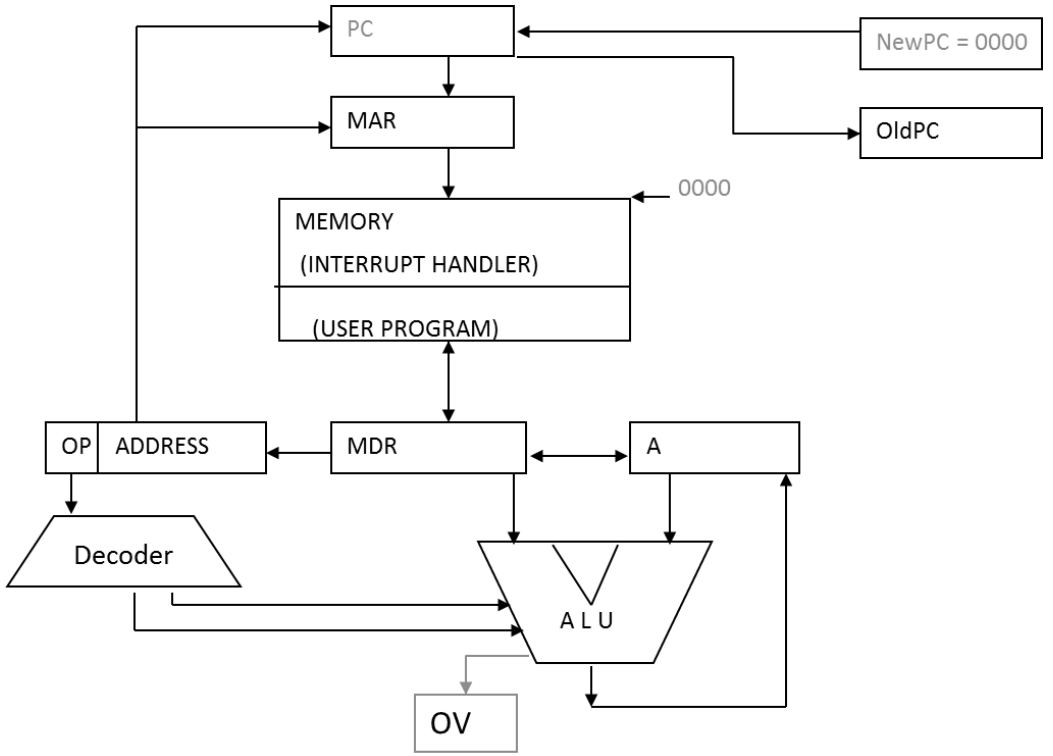
To do this, instead of halting, we just load the PC with the start address of the interrupt handler. For now, we'll call that **NewPC**.

F Handler (hidden instruction)

If OV=1

then PC ← NewPC

DECODER ← 00



Interrupt Handlers: Implications

Along with LOAD, STORE, the I/O instructions, and the fetch cycle, the interrupt handler serves as part of a bridge between hardware and software.

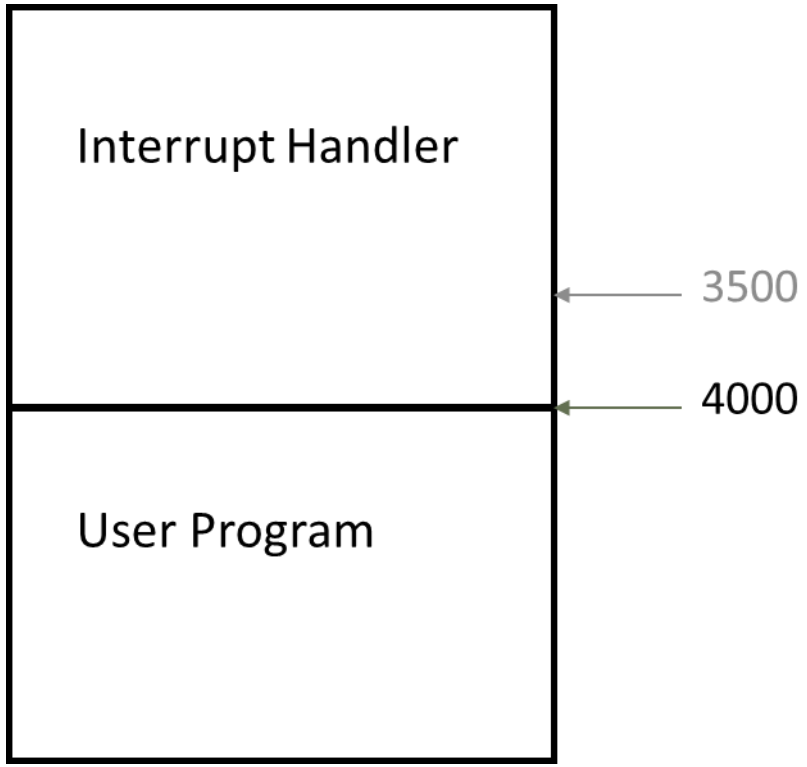
- At least as importantly, with a little more work, it can serve as a bridge between *user* software and *system* software
- *The interrupt handler is one of the first steps toward a modern operating system*
- But before we can do much with it, we'd better put some walls around it

Shared Memory

Nothing executes unless it's in memory, and that includes the interrupt handler.

So let's take a simple example.

- The interrupt handler occupies spaces 3000-3999
- The user program begins at space 4000
- What happens if the user writes to space 3500?



Shared Memory

Nothing executes u
that includes the i

So let's take a simp

- The interrupt har
  3999
- The user program
- What happens if

BAD THINGS

3500

4000

Shared Memory

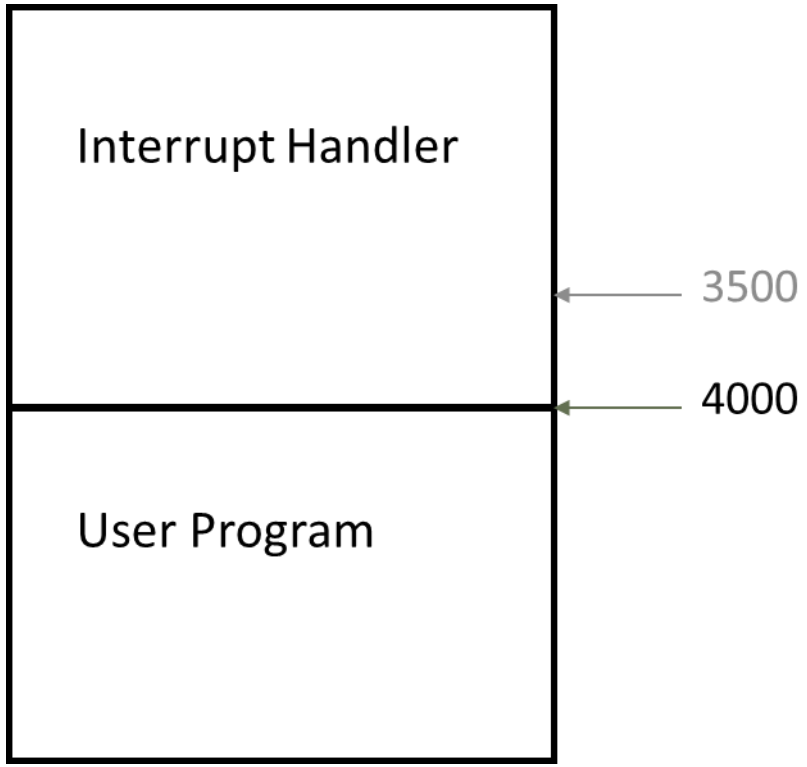
Nothing executes unless it's in memory, and that includes the interrupt handler.

So let's take a simple example.

- The interrupt handler occupies spaces 3000-3999
- The user program begins at space 4000
- What happens if the user writes to space 3500?

We need a way to protect the interrupt handler routine from user programs stepping on it.

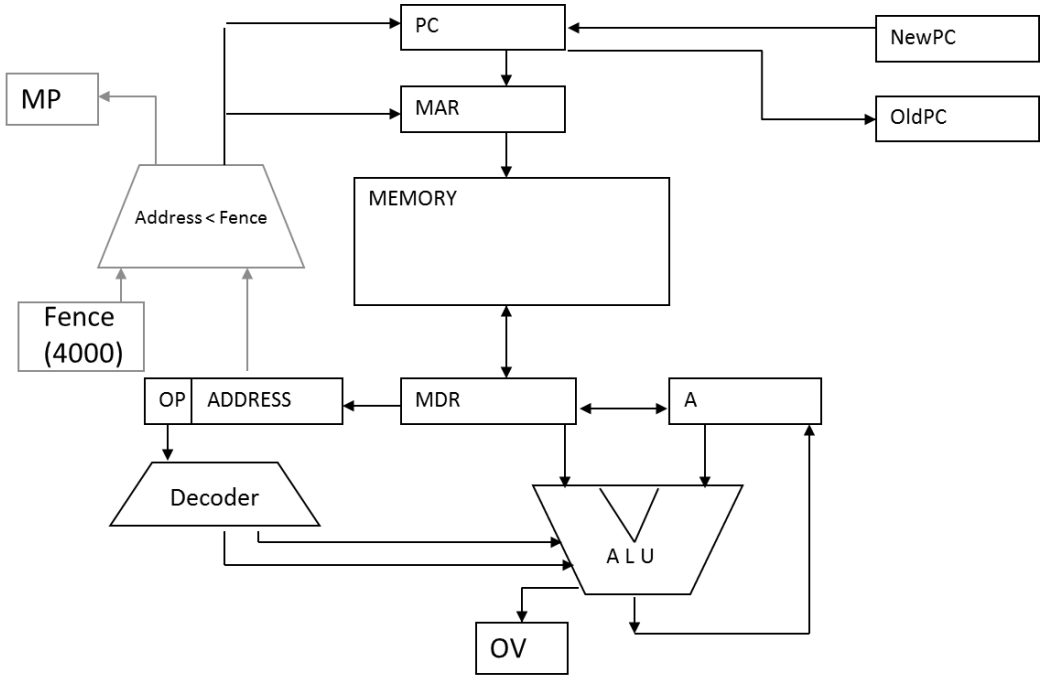
- In fact, it'd be great if we could protect *all* programs from *all other* programs stepping on them
- But let's take this a step at a time



Simple Memory Protection

Our memory protection scheme has three components:

- A **Fence Register** loaded with the address of the boundary between the interrupt handler routine and the user program
- An **Address Comparator** that compares the fence register with any addresses that the user program attempts to access
- A **flip-flop** that is set to 1 if a user program tries to access something it shouldn't



Instruction Changes

0 Fetch (hidden instruction)

MAR ←PC

MDR ←MEM[MAR]

IR ←MDR

PC ←PC+1

DECODER ←IR.OP

2 ADD

MAR←IR.Address

MDR ←MEM[MAR]

A ← A + MDR

DECODER ←0F

1 LOAD

MAR←IR.Address

IF MP=0 then

MDR ←MEM[MAR]

A ← MDR

DECODER←0F

3 STORE

MAR←IR.Address

IF MP=0 then

MDR ←A

MEM[MAR] ←MDR

DECODER ←0F

7 HALT

Now that we've got memory protection hardware, we need to use it.

- LOAD and STORE need to honor it
- The handler needs to react to a violation
- We treat a violation as an interrupt

F Handler (hidden instruction)

If OV=1 then PC ← NewPC

If MP=1 then PC ← NewPC

DECODER ← 00

Still Not Done

- …but *what if a user program modifies the fence register?*
  - (Yes, programmers will, in fact, do that if we let them.)
    - (Really.)

Privileged Instructions & System Modes

We've protected memory, and that's all well and good… but *what would happen if a user program tried to modify the memory fence register?*

- The register isn't in memory – memory protection doesn't help it
- We need the idea of *privileged instructions* – instructions that user programs cannot access

To distinguish between times when privileged instructions are allowed and not, the computer operates in two **modes.**

- User mode: 0
- Supervisor mode: 1

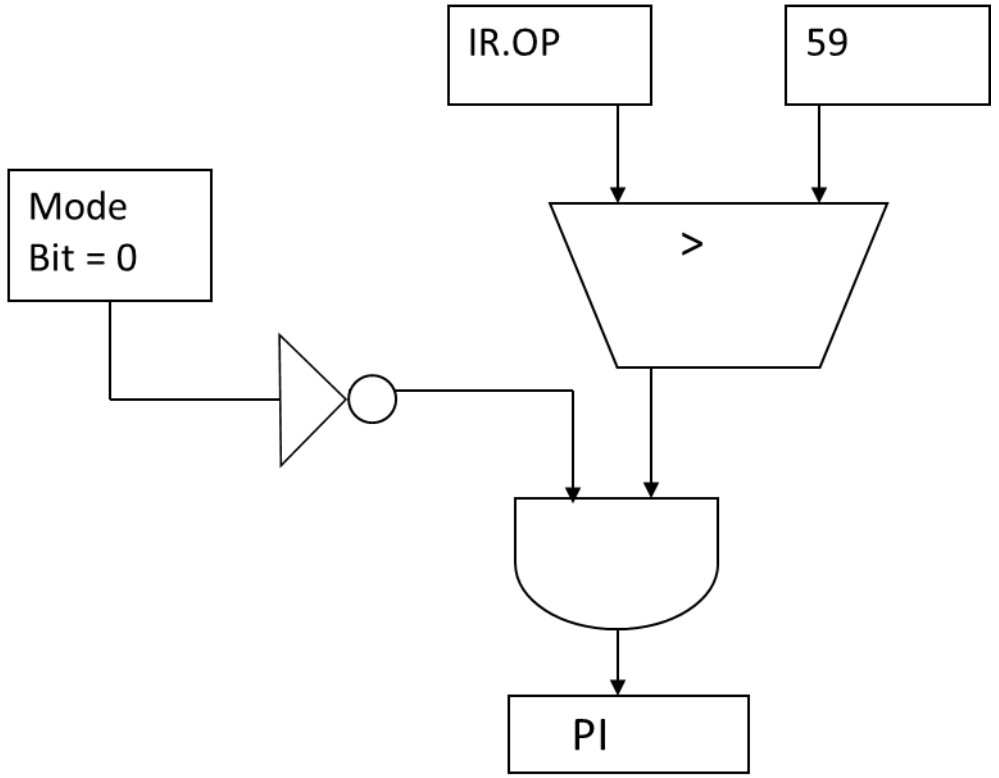
Supervisor mode is a mode exclusively reserved for the interrupt handler.

- In fact, *supervisor* is another name for the interrupt handler
- In user mode, only a subset of the instruction set can be used
- The supervisor has access to all instructions

Privileged Instructions: Implementation

We have to add a few more pieces to the CPU.

- First, another flip-flop: the *mode bit*
- Second, another comparator
  - We organize the instruction set so that all and only privileged instructions have opcodes higher than some *n*
  - Here *n* = 59
  - The comparator compares the opcode with *n*
- If the comparator finds that IR.OP > *n*, then the instruction is privileged
- If the instruction is privileged *and* the mode bit is 0 – indicating user mode – then we have an invalid instruction
  - And – you guessed it – we call the interrupt handler



Types of Interrupts

There are several types of interrupts. So far we've been referring to ones created by errors – such as an overflow, memory address violation, or invalid instruction.

- Whenever one of these occurs, the supervisor *will* abort the process

Obviously, that's not the only type of interrupt.

- **I/O interrupts** occur when a device sends a signal to inform the CPU that an I/O operation has been completed
- **External** interrupts occur when non-I/O system components – such as the timer – send signals to the CPU
- **System calls** are interrupts generated *intentionally* by user processes, to request services from the operating system

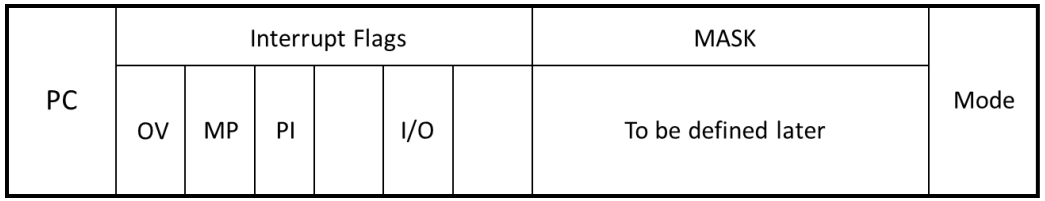
When interrupts other than errors occur, the state of the running process must be saved before the supervisor services them.

Program State Word (PSW)

The PSW **Program State Word** is a structure in the CPU that holds information about the state of a program.

In this structure, we have (at least):

- The program counter
- The interrupt flags we've discussed
  - OV (Overflow)
  - MP (Memory Protection)
  - PI (Privileged Instruction)
  - I/O
- The mask, which we'll define later
- The system mode



Instruction F with All Four Flags (so far)

```
IF OV = 1 THEN              /* Overflow */
  PC ← NEWPC; MODE ← 1      /* ABEND */
ELSE IF MP = 1 THEN         /* Memory violation */
  PC ← NEWPC; MODE ← 1      /* ABEND */
ELSE IF PI = 1 THEN         /* Invalid instruction */
  PC ← NEWPC; MODE ← 1      /* ABEND */
ELSE IF I/O = 1 THEN        /* I/O interrupt */
  OLDPC ← PC
  /* Perform additional functions to save process state here */
  PC ← NEWPC
  MODE ← 1
END IF
DECODER ← 00
```

More on System Calls

The supervisor can use both user and privileged instructions; user programs can't.

- This means a user program can't execute even simple system calls like `open()` or `read()` by itself
- Communication with the supervisor is required
- This is the purpose of software-generated interrupts, or system calls

In a system call, the user program voluntarily relinquishes control to the supervisor.

- The supervisor then services the interrupt generated by the system call
- The interrupt handler code, before returning control to the user program, will generally copy any information the program requested into memory space accessible to it

Next Time: Program State and System Calls