Lecture 2

Interrupt Handling

by

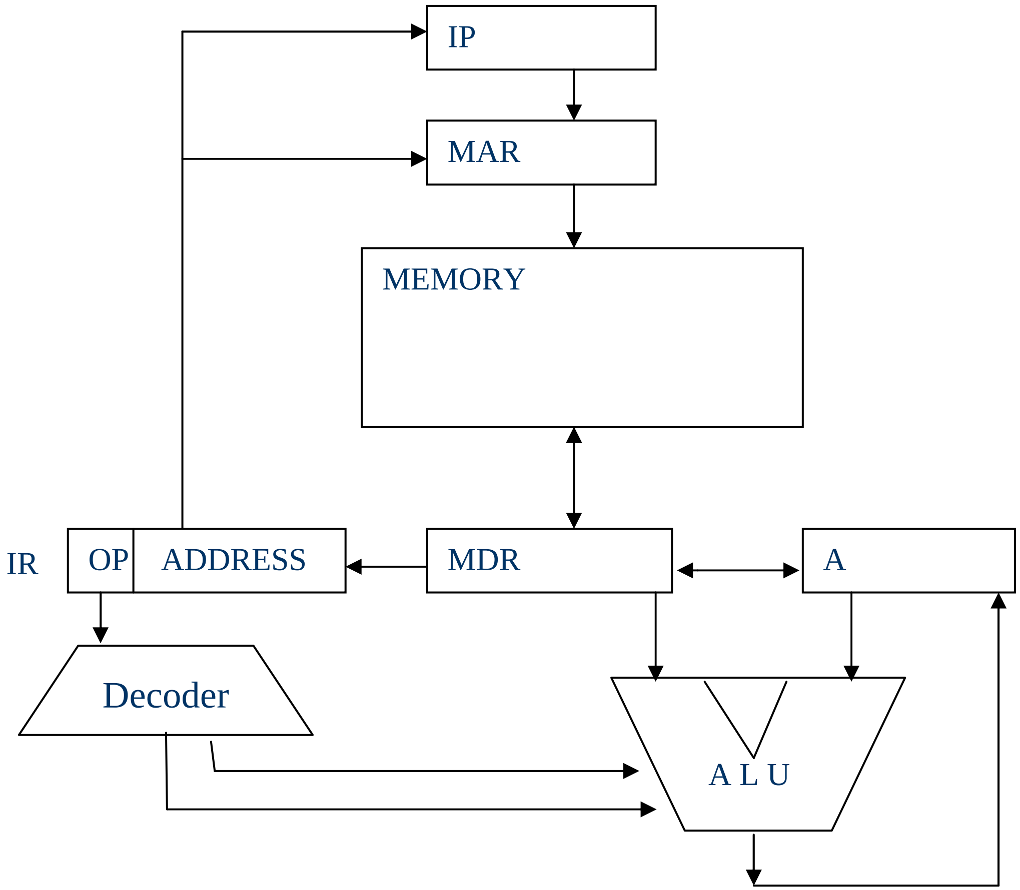
Euripides Montagne

University of Central Florida

Outline

1. The structure of a tiny computer.
2. A program as an isolated system.
3. The interrupt mechanism.
4. The hardware/software interface.
5. Interrupt Types.

Von-Neumann Machine (VN)



Instruction Cycle

- Instruction cycle, or machine cycle, in VN is composed of 2 steps:
- 1. Fetch Cycle: instructions are retrieved from memory
- 2. Execution Cycle: instructions are executed
- A hardware description language will be used to understand how instructions are executed in VN

Definitions

- IP: Instruction Pointer is a register that holds the address of the next instruction to be executed.
- MAR: Memory Address Register is used to locate a specific memory location to read or write its content.
- MEM: Main storage, or RAM (Random Access Memory) and is used to store programs and data.

Definition of MDR

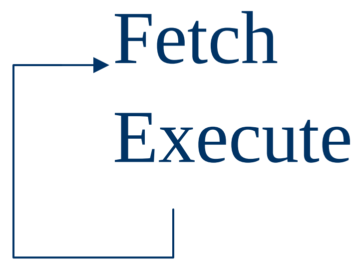
MDR: Memory Data Register is a bi-directional register used to receive the content of the memory location addressed by MAR or to store a value in a memory location addressed by MAR. This register receives either instructions or data from memory

Definitions Cont.

- IR: Instruction Register is used to store instructions
- DECODER: Depending on the value of the IR, this device will send signals through the appropriate lines to execute an instruction.
- A: Accumulator is used to store data to be used as input to the ALU.
- ALU: Arithmetic Logic Unit is used to execute mathematical instructions such as ADD, or MULTIPLY

Fetch Execute Cycle

- In VN, the instruction cycle is given by the following loop:

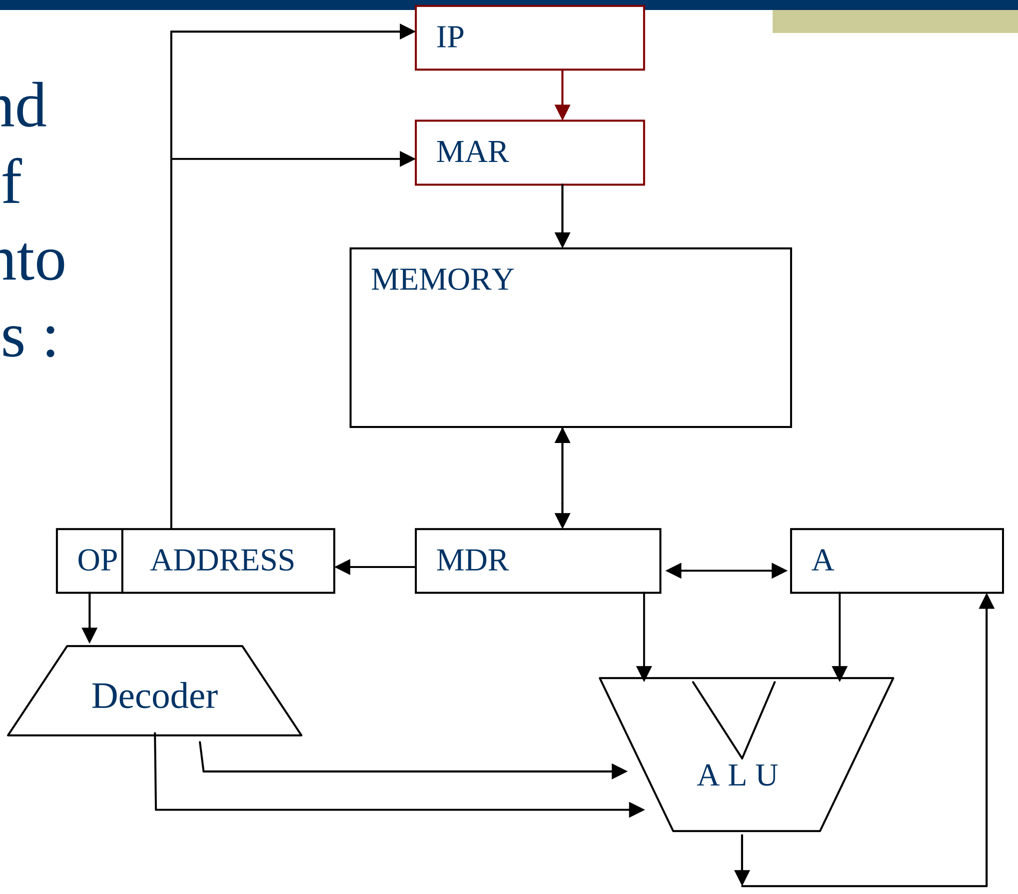


- In order to explain further details about the fetch /execute cycle, the data movements along different paths can be described in 4 steps.

Data Movement 1

- Given register IP and MAR the transfer of the contents of IP into MAR is indicated as :

  MAR ← IP

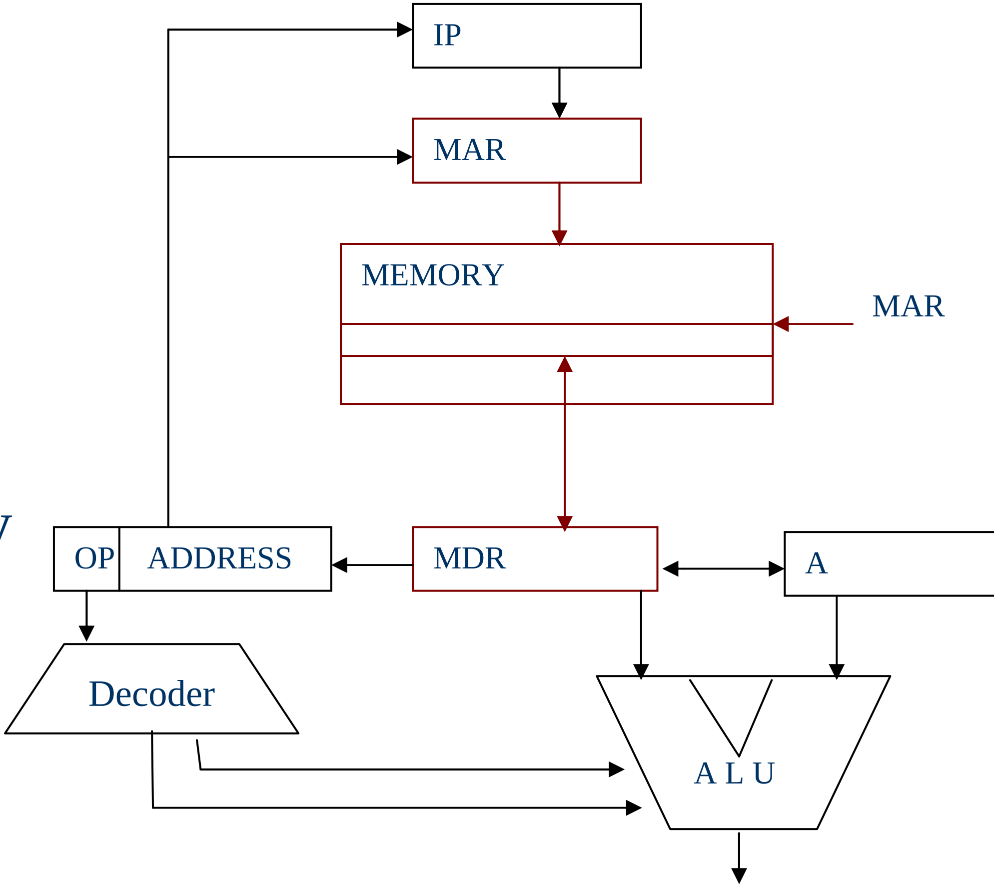


Data Movement 2

- To transfer information from a memory location to the register MDR, we use:

  MDR ← MEM[MAR]

- The address of the memory location has been stored previously into the MAR register



Data Movement 3

- To transfer information from the MDR register to a memory location, we use:

  MEM [MAR] ← MDR

  *see previous slide for diagram

- The address of the memory location has been previously stored into the MAR

Instruction Register Properties

- The Instruction Register (IR) has two fields: Operation (OP) and the ADDRESS.
- These fields can be accessed using the selector operator "."

Data Movement 4

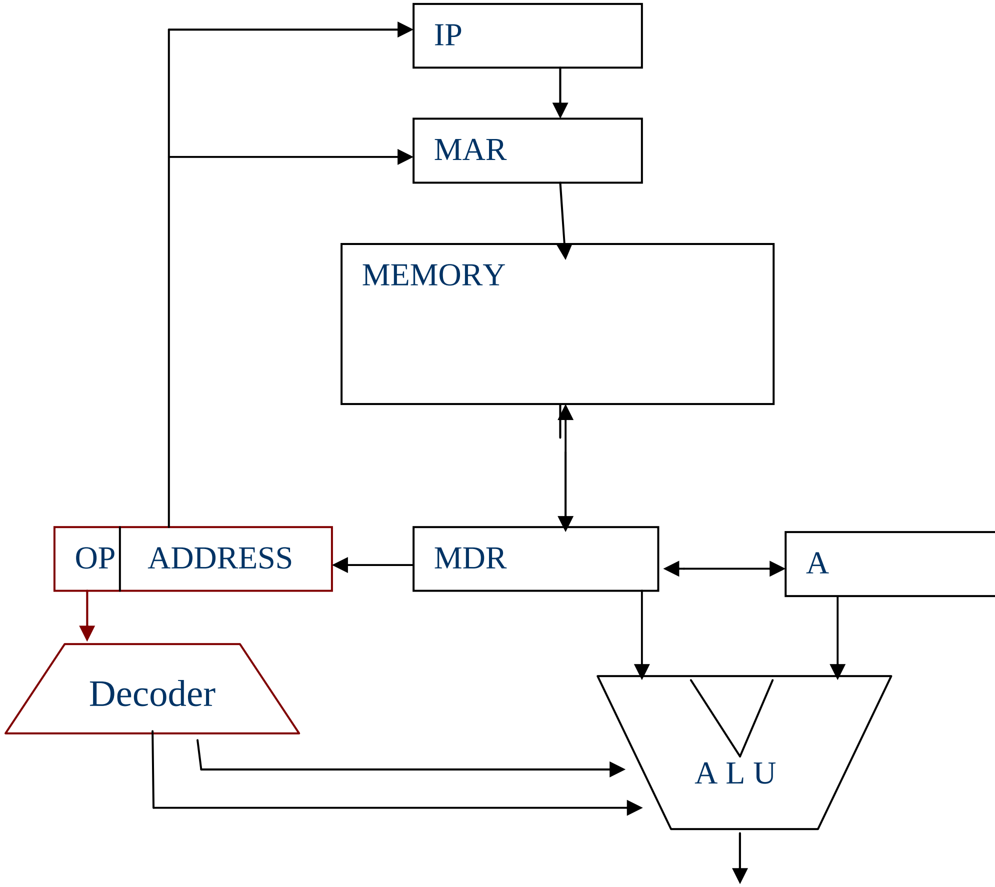
- The operation field of the IR register is sent to the DECODER as:

  DECODER ← IR.OP

- The Operation portion of the field is accessed as IR.OP
- DECODER: If the value of IR.OP==0, then the decoder can be set to execute the fetch cycle again.

Data Movement 4 Cont.

DECODER ← IR.OP



Instruction Cycle

- The instruction cycle has 2 components.
- Fetch cycle retrieves the instruction from memory.
- Execution cycle carries out the instruction loaded previously.

00 Fetch Cycle

1. MAR ← IP
2. MDR ← MEM[MAR]
3. IR ← MDR
4. IP ← IP+1
5. DECODER ← IR.OP

1. Copy contents of IP into MAR
2. Load content of memory location into MDR
3. Copy value stored in MDR into IR
4. Increment IP register
5. Select Instruction to be executed

Execution: 01 LOAD

1. MAR ← IR.ADDR
2. MDR ← MEM[MAR]
3. A ← MDR
4. DECODER ← 00

1. Copy the IR address value field into MAR
2. Load the content of a memory location into MDR
3. Copy content of MDR into A register
4. Set Decoder to execute Fetch Cycle

Execution: 02 ADD

1. MAR ← IR.ADDR
2. MDR ← MEM[MAR]
3. A ← A + MDR
4. DECODER ← 00

1. Copy the IR address value field into MAR
2. Load content of memory location to MDR
3. Add contents of MDR and A register and store result into A
4. Set Decoder to execute Fetch cycle

Execution: 03 STORE

1. MAR ← IR.ADDR
2. MDR ← A
3. MEM[MAR] ← MDR
4. DECODER ← 00

1. Copy the IR address value field into MAR
2. Copy A register contents into MDR
3. Copy content of MDR into a memory location
4. Set Decoder to execute fetch cycle

Execution: 04 END

1. STOP

1. Program ends normally

Instruction Set Architecture

00 Fetch (hidden instruction)

MAR ← IP

MDR ← MEM[MAR]

IR ← MDR

IP ← IP+1

DECODER ← IR.OP

02 Add

MAR ← IR.Address

MDR ← MEM[MAR]

A ← A + MDR

DECODER ← 00

01 Load

MAR ← IR.Address

MDR ← MEM[MAR]

A ← MDR

DECODER ← 00

03 Store

MAR ← IR.Address

MDR ← A

MEM[MAR] ← MDR

DECODER ← 00

04 Stop

One Address Architecture

- The instruction format of this one-address architecture is:

  operation<address>

- Address are given in hexadecimal and are preceded by an “x”, for instance x56

Example One-Address Program

- Memory Address

x20	450
x21	300
x22	750 (after program execution)
x23	Load <x20>
x24	Add <x21>
x25	Store<x22>
x26	End

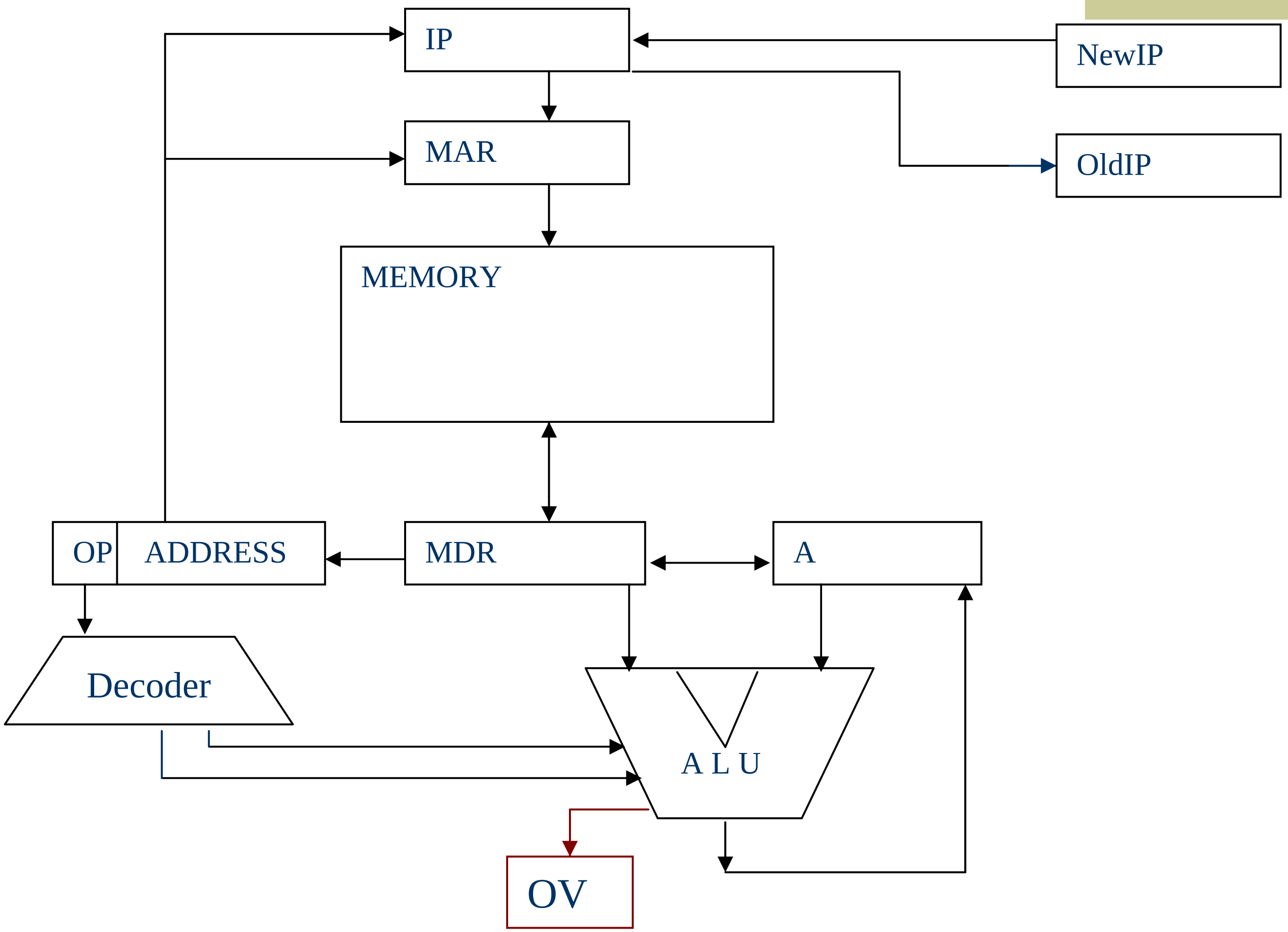
Programs with Errors

- So far, we have a computer that can execute programs free from errors.
- What would happen if an overflow occurred while executing an addition operation?
- We need a mechanism to detect this type of event and take appropriate actions.

Overflow Detection

- A flip/flop will be added to the ALU for detecting overflow
- The Fetch/Execute cycle has to be extended to: Fetch/Execute/Interrupt cycle.
- An abnormal end (ABEND) has to be indicated.

VN with Overflow Flip/Flop



Interrupt Cycle

- In the interrupt cycle, the CPU has to check for an interrupt each time an instruction is executed.
- Modifications have to be made to the instruction set to incorporate the interrupt cycle.
- An operation code of 05 will be added to accommodate the Interrupt Cycle.
- At the end of each execution cycle, the DECODER will be set to 05 instead of 00, to check for interrupts at the end of each execution cycle.

Interrupt Cycle 05

1. If OV=1

   Then HALT

   DECODER ← 00

1. Abnormal End (ABEND) for Overflow
2. Set Decoder to Fetch Cycle

ISA –Interrupt cycle

01 Load

MAR ← IR.Address

MDR ← MEM[MAR]

A ← MDR

DECODER ← 05

02 Add

MAR ← IR.Address

MDR ← MEM[MAR]

A ← A + MDR

DECODER ← 05

03 Store

MAR ← IR.Address

MDR ← A

MEM[MAR] ← MDR

DECODER ← 05

04 Stop

05 Abend

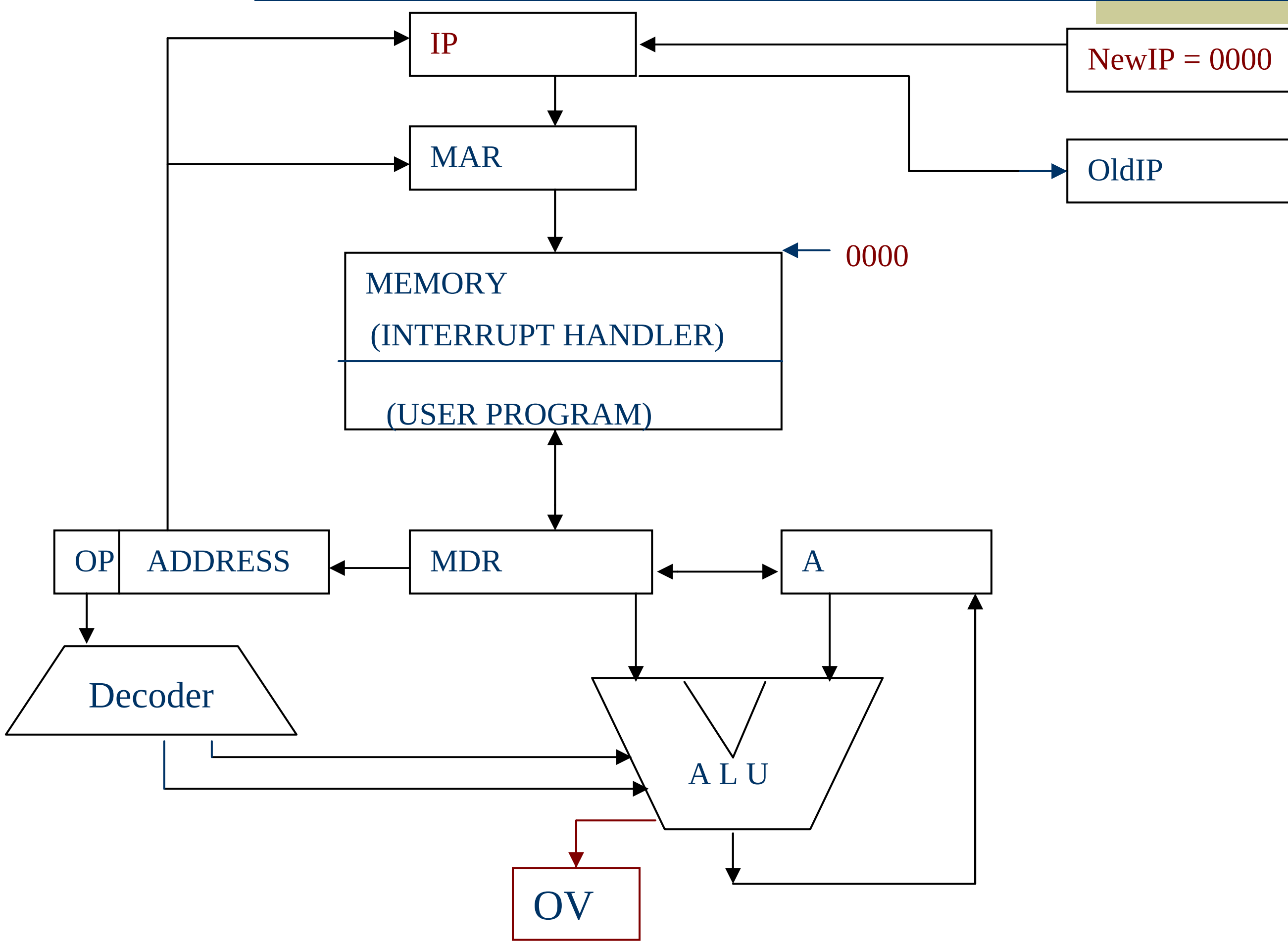
IF OV = 1 Then HALT

DECODER ← 00

Interrupt Handling Routine

- Instead of halting the machine, the flow of execution can be transferred to an *interrupt handling routine*
- This is done by loading the IP register with the start address of the interrupt handler in memory from NEWIP.
- Causes a change in the Interrupt Cycle

Interrupt Handler Takes Control of VN



05 Interrupt Cycle

If OV=1

Then IP ← NEWIP

DECODER ← 00

- Jump to interrupt handler at memory location 1000
- Set decoder to fetch cycle

Hardware/Software Bridge

01 Load

- MAR ← IR.Address
- MDR ← MEM[MAR]
- A ← MDR
- DECODER ← 05

02 Add

- MAR ← IR.Address
- MDR ← MEM[MAR]
- A ← A + MDR
- DECODER ← 05

03 Store

- MAR ← IR.Address
- MDR ← A
- MEM[MAR] ← MDR
- DECODER ← 05

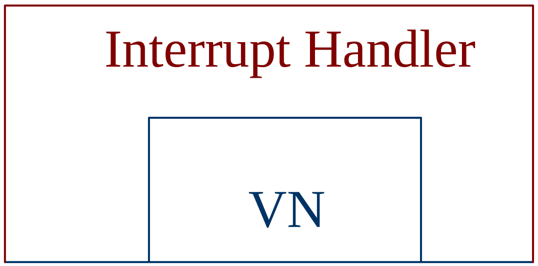
04 Stop

05 Interrupt Handler Routine

- IF OV = 1 IP ← NEWIP
- DECODER ← 00

Virtual Machine

- The interrupt handler is the first extension layer or virtual machine developed over VN
- First step towards an operating system

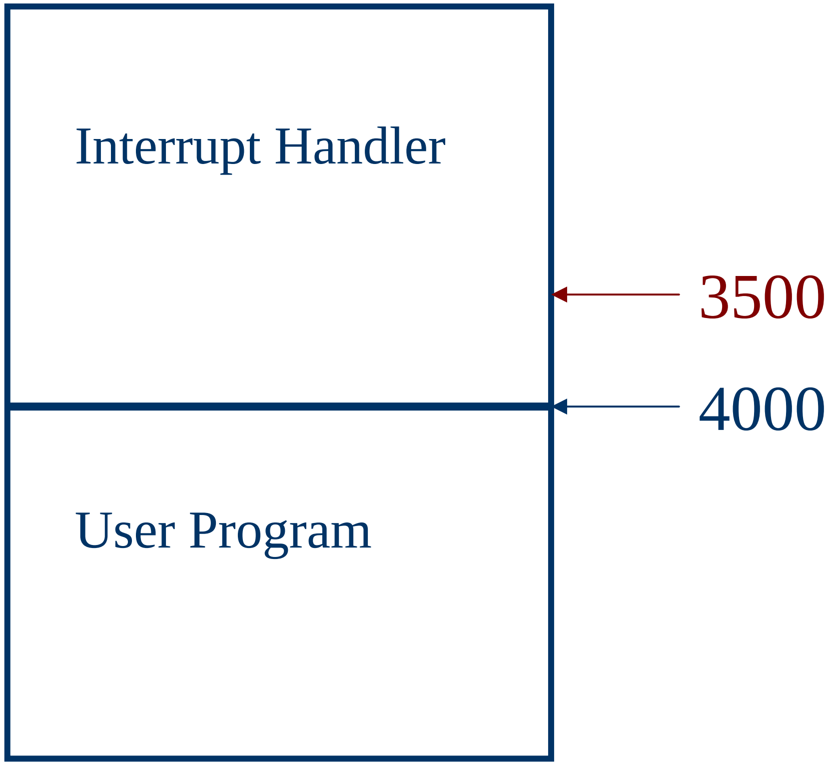


Interrupt Handler Virtual Machine

Shared Memory

- The interrupt handler has to be loaded into memory along with any user program.
- Sharing memory space raises a new problem: the user program might eventually execute an instruction which may modify the interrupt handler routine

Shared Memory Example



Interrupt Handler is loaded at MEM[0] with a length of 4000 words.

User program executes:

STORE<3500>, thus modifying the handler routine.

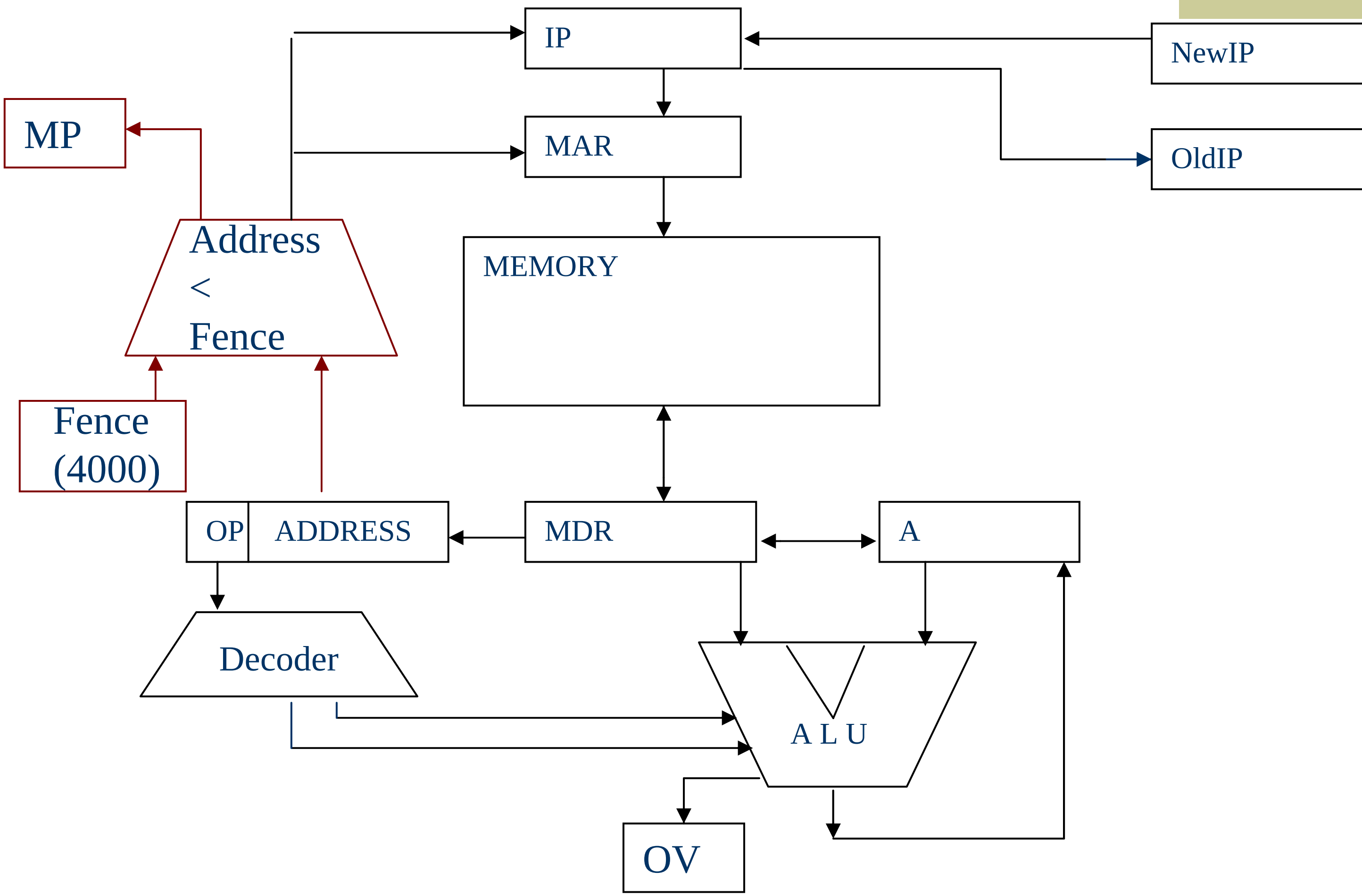
Memory Protection

- A new mechanism must be implemented in order to protect the interrupt handler routine from user programs.
- The memory protection mechanism has three components: a fence register, a device to compare addresses, and a flip flop to be set if a memory violation occurs.

Memory Protection Components

- Fence Register: register loaded with the address of the boundary between the interrupt handler routine and the user program
- Device for Address Comparisons: compares the fence register with any addresses that the user program attempts to access
- Flip/Flop: is set to 1 if a memory violation occurs

VN with Memory Protection



Changes to the ISA

- With the inclusion of the mechanism to protect the Interrupt Handler, some modifications need to be made to the ISA (Instruction Set Architecture)
- Instructions Load, Add, and Store have to be modified to check the value of the Memory Protection (MP) once the first step of those instructions has executed

Modified ISA

01 Load

MAR ← IR.Address

If MP=0 Then

MDR ← MEM[MAR]

A ← MDR

DECODER ← 05

02 Add

MAR ← IR.Address

If MP=0 Then

MDR ← MEM[MAR]

A ← A + MDR

DECODER ← 05

03 Store

MAR ← IR.Address

If MP=0 Then

MDR ← A

MEM[MAR] ← MDR

Decoder ← 05

05 Interrupt Handler Routine

IF OV = 1 IP ← NEWIP

IF MP = 1 IP ← NEWIP

DECODER ← 00

Program State Word (PSW)

- The PSW, or Program State Word, is a structure that give us information about the state of a program.
- In this register, we have the IP, MODE, Interrupt Flags, and the Mask(defined later)

Program State Word

IP	Interrupt Flags						MASK	
	OV	MP					To be defined later	

Privileged Instructions

- What if a user program attempted to modify the fence register?

  The register is not protected so it does not fall under the previous memory protection mechanism.

- Use the idea of privileged instructions to denote which instructions are prohibited to user programs

Privileged Instruction Implementation

- To distinguish between times when privileged instructions either are or are not allowed, the computer operates in two *modes*
- User mode: 0
- Supervisor mode: 1
- From now on, *interrupt handler* and *supervisor* are terms that can be used interchangeably
- In User mode, only a subset of the instruction set can be used
- The supervisor has access to all instructions

Implementing Privileged Instructions cont.

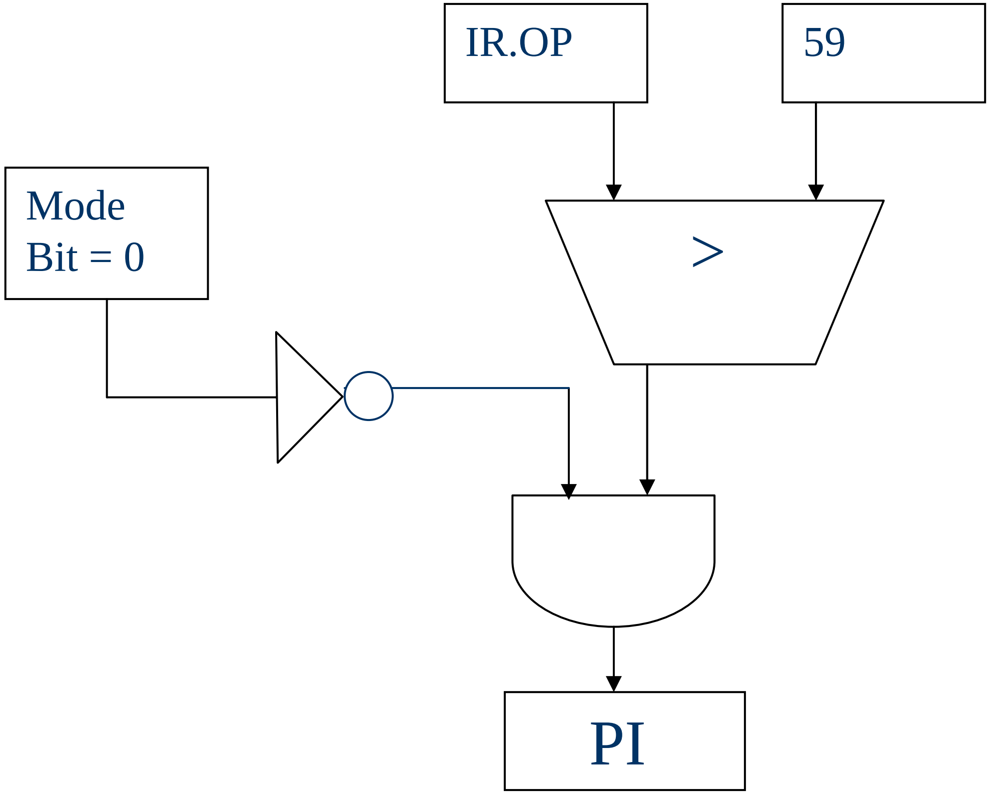
- 1. Add another flip/flop (flag) to the CPU and denote it as the mode bit
- 2. Create a mechanism in the CPU to avoid the execution of privileged instructions by user programs
- 3. The instruction set has to be organized in such a way that all privileged instructions have operation codes greater than a given number.

  -For example, if the ISA has 120 instructions, privileged instructions will have operation codes greater than 59

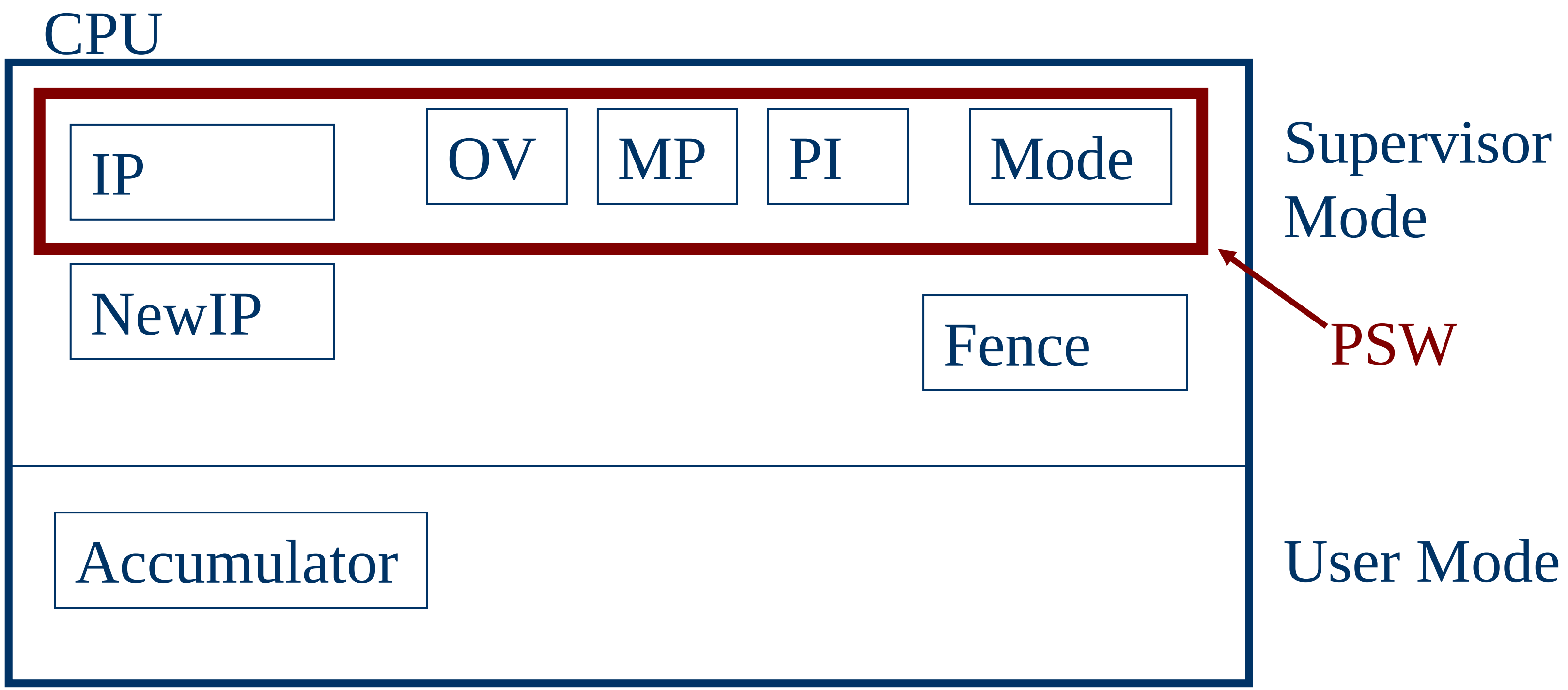
Mechanism for User/Supervisor Modes

- This device compares the opcode in the Instruction Register (IR.OP) with the opcode of the last non-privileged instruction.
- If the outcome yields a “1”, then this is a privileged instruction.
- This outcome is then compared with the mode bit.
- If the mode is 0 (indicating user mode), and it is a privileged instruction, then the Privileged Instruction bit (PI) is set to one.
- The hardware will detect the event, and the interrupt handler routine will be executed

Mechanism for User/Supervisor Modes Cont.



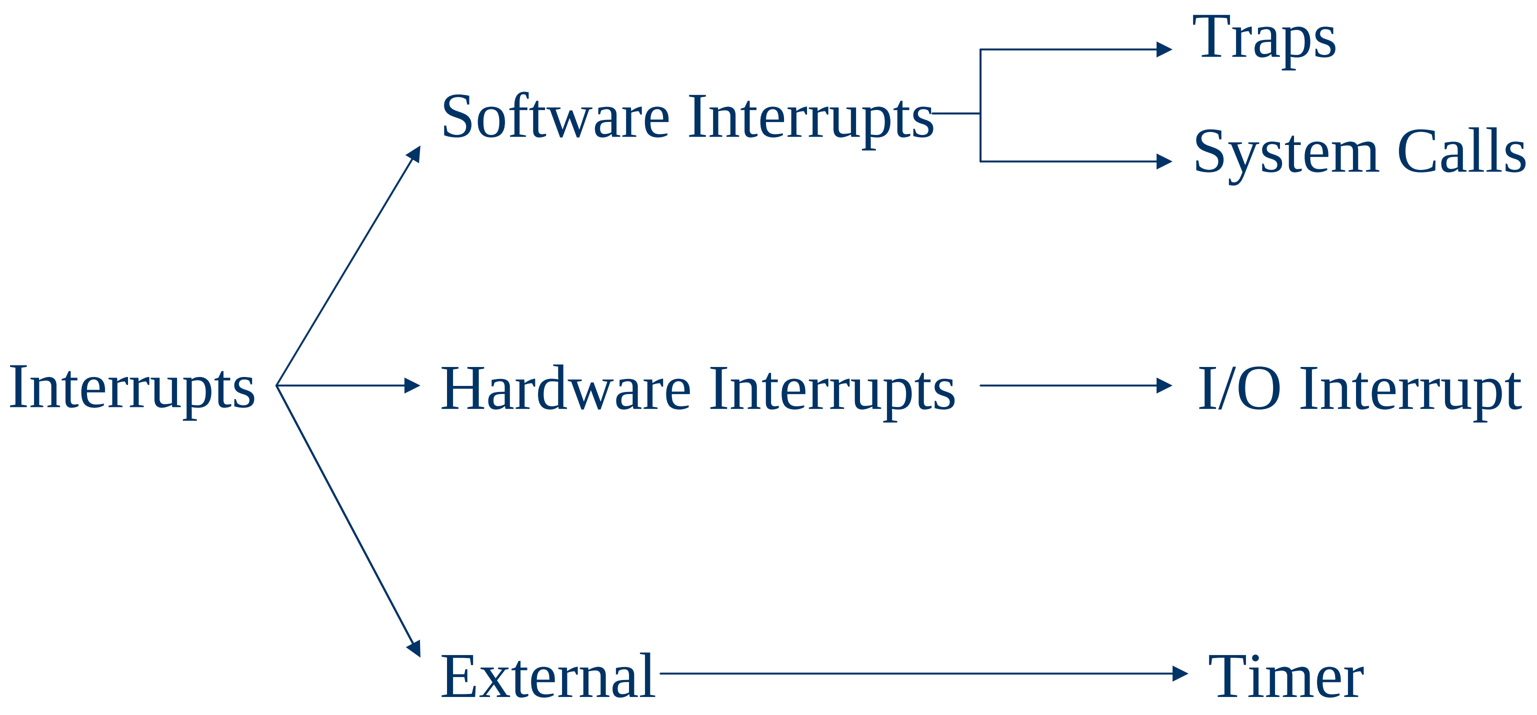
CPU After Mode Flag Addition



PSW After Mode and PI flag Addition

IP	Interrupt Flags						MASK	Mode
	OV	MP	PI				To be defined later	

Types of Interrupts



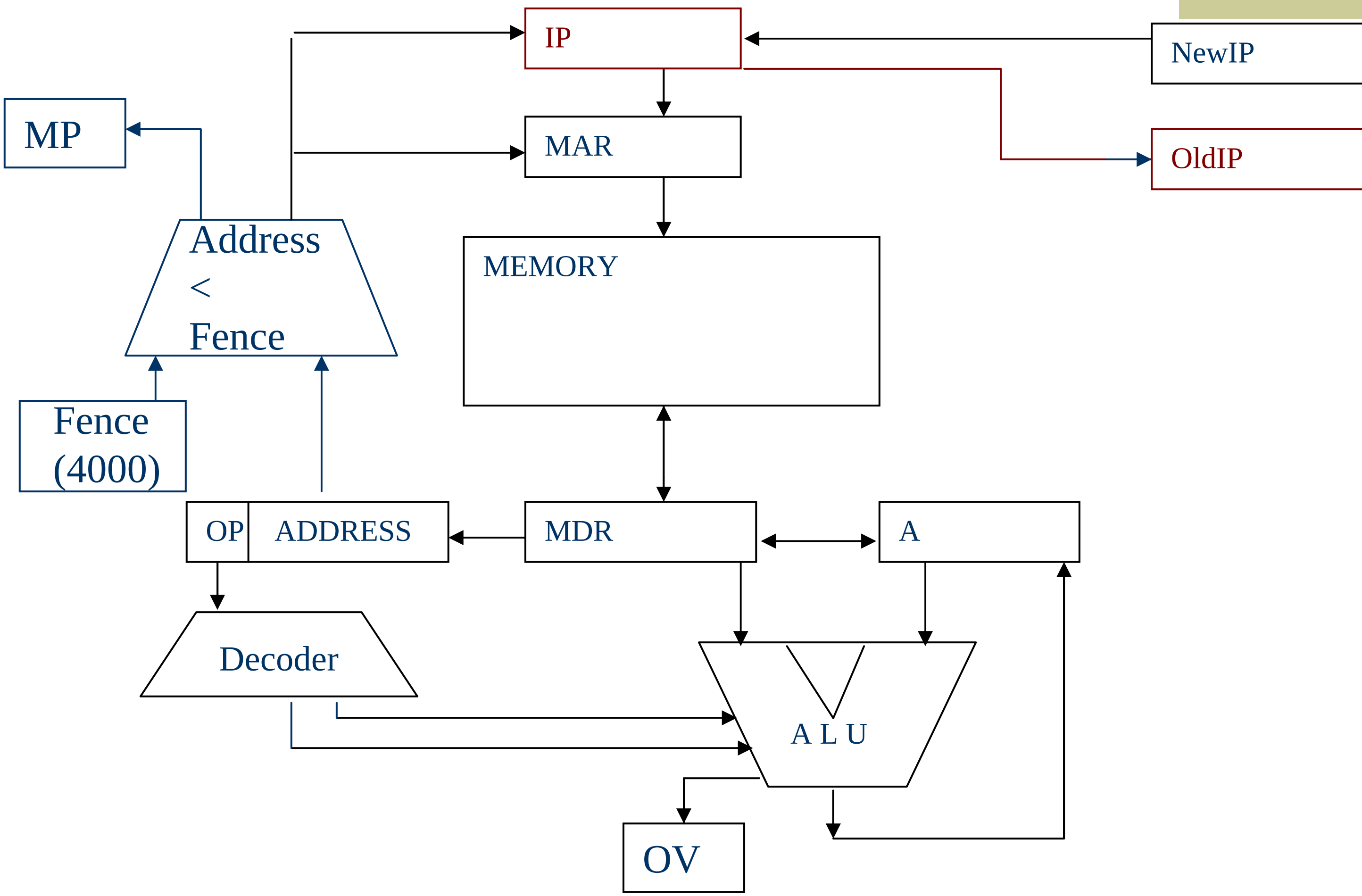
Traps

- An interrupt is an exceptional event that is automatically handled by the interrupt handler.
- In the case of an overflow, memory addressing violation, and the use of privileged instruction in user mode, the handler will abort the program
- These types of interrupts are called *traps*
- All traps are going to be considered synchronous interrupts

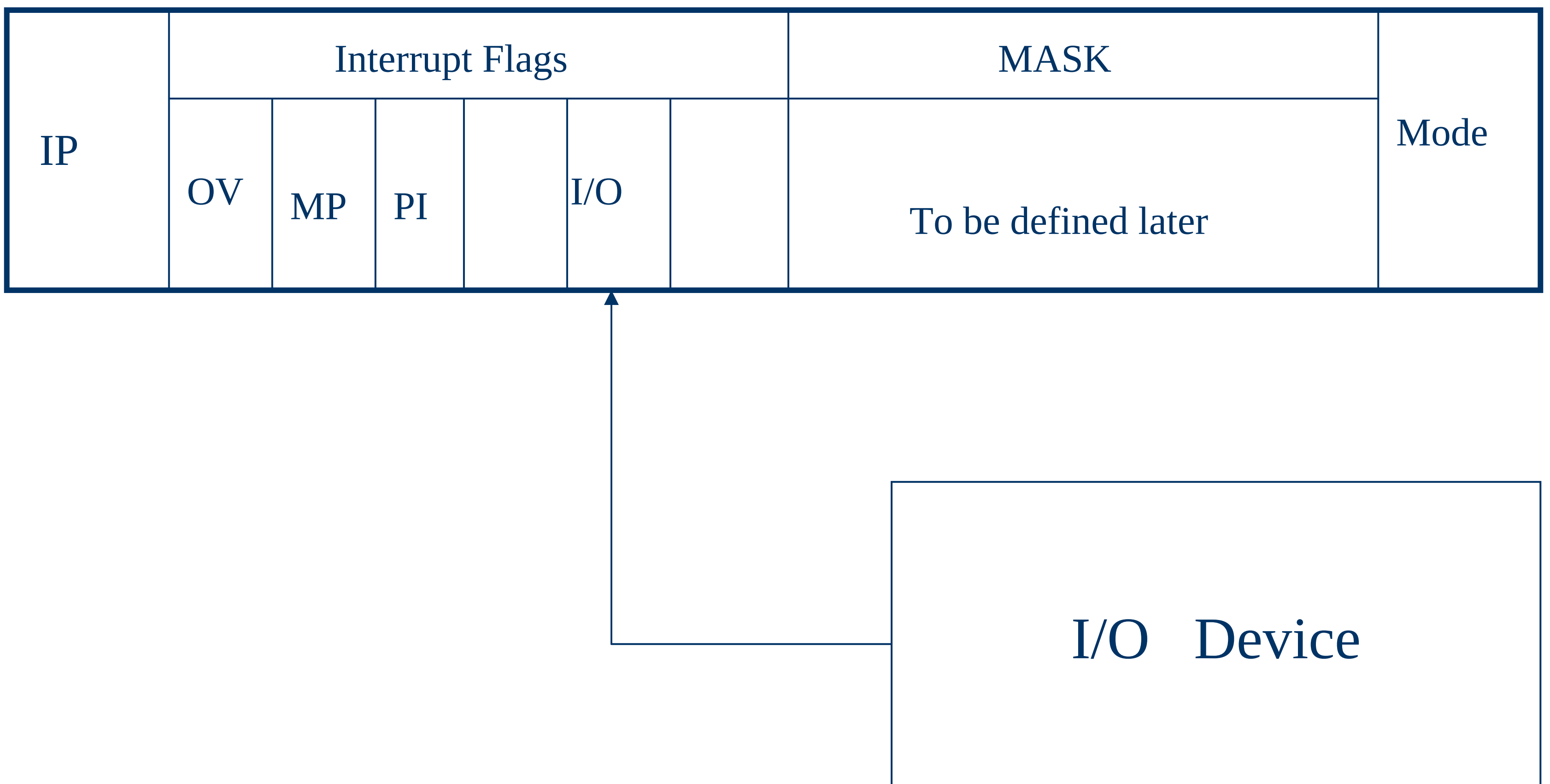
I/O Interrupts

- This type of interrupt occurs when a device sends a signal to inform the CPU that an I/O operation has been completed
- An I/O flag is used to handle this type of interrupt
- When an I/O interrupt occurs, the Program State of the running program is saved so that it can be restarted from the same point after the interrupt has been handled.

Saving the state of the running program



Program State Word



05 Interrupt Cycle

IF OV = 1 THEN IP ← NEWIP; MODE ← 1 (ABEND).

IF MP = 1 THEN IP ← NEWIP; MODE ← 1 (ABEND).

IF PI = 1 THEN IP ← NEWIP; MODE ← 1 (ABEND)

IF I/O = 1 THEN OLDIP← IP;
IP ←NEWIP;
MODE←1;

DECODER ← 00

Supervisor

- The Supervisor can use both user and privileged instructions.
- Sometimes a user program requires some services from the Supervisor, such as opening and reading files.
- A program cannot execute open or read functions itself, and so needs a mechanism to communicate with the Supervisor

SuperVisorCall (SVC)

- An SVC is also known as a System Call
- It is a mechanism to request service from the Supervisor or OS.
- This mechanism is a type of interrupt, called a *software interrupt* because the program itself relinquishes control to the Supervisor as part of its instructions.

System Calls

- There are two types of system calls:

  1. Allows user programs to ask for service (instructions found below opcode 59)

  2. Privileged Instructions (over opcode 59)

SCVT

- The System Call Vector Table(SCVT) contains a different memory address location for the beginning of each service call
- Service calls are actually programs because they require multiple instructions to execute
- Each memory address contained in the SCVT points to runtime library, generally written in assembly language, which contains instructions to execute the call

Runtime Libraries

- Runtime Libraries: precompiled procedures that can be called at runtime
- Runtime Libraries set a new flip/flop, called the SVC flag, to “1”, which causes the system to switch to Supervisor Mode in the Interrupt Cycle

Properties of Runtime Libraries

- Libraries are shared by all programs
- Are not allowed to be modified by any program.

SVC Instruction Format

- SVC(index) is the format for system calls.
- The index is the entry point in the SCVT

Read → Compiler → SVC(index) (IR.OP=SVC, IR.ADDR=index)

80 SVC(index)

80 SVC(index)

OLDIP ← IP;

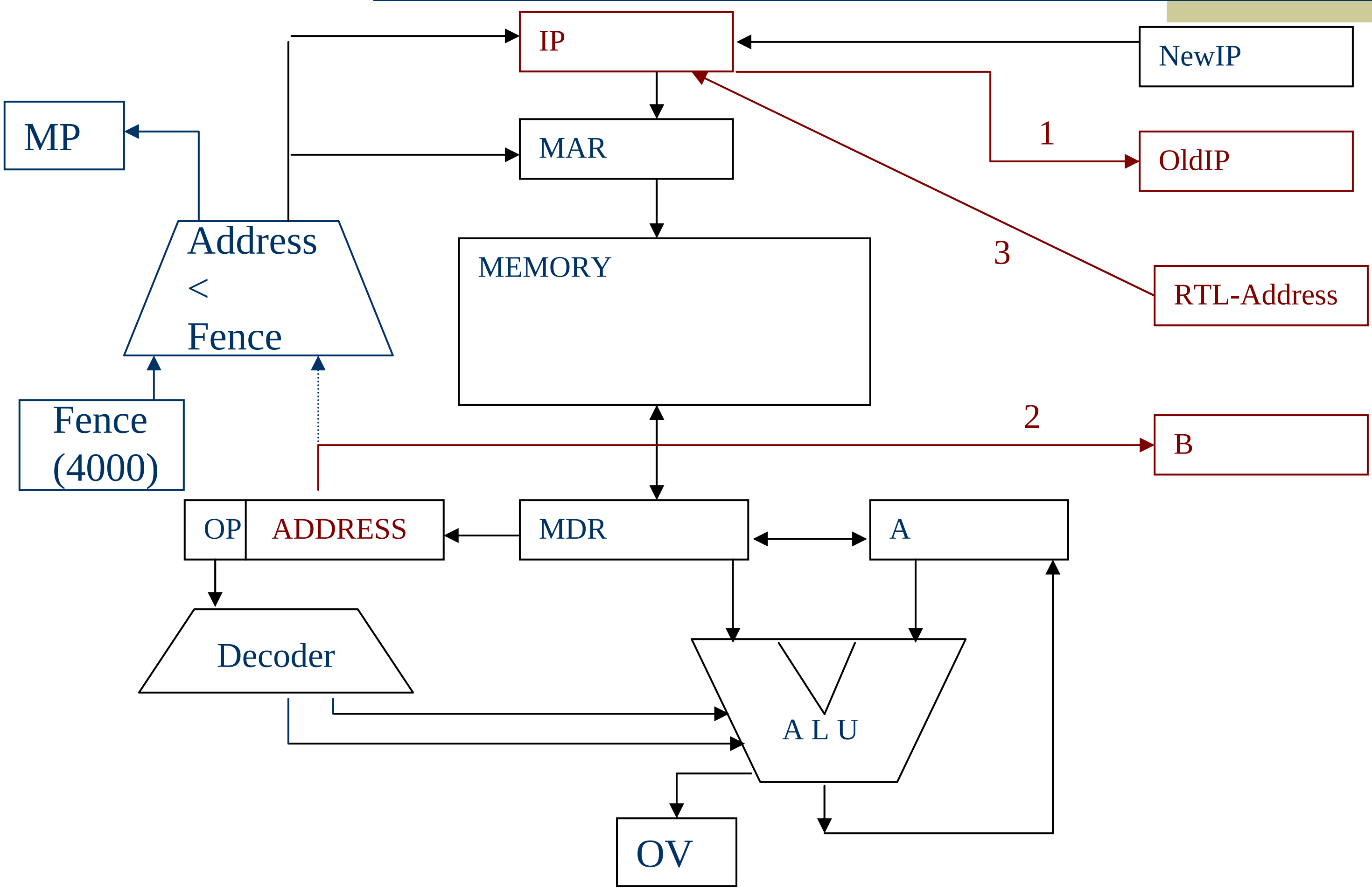
B ← IR.ADDRESS

IP ← RTL-ADDRESS

DECODER ← 05

- Save IP of current program
- The Index value is temporarily loaded into register B
- Address of Runtime Library
- Transfer to Interrupt Cycle

SVC(read) = 80(4)


IP
NewIP
MP
MAR
1
OldIP
Address
<
Fence
MEMORY
3
RTL-Address
Fence
(4000)
2
B
OP
ADDRESS
MDR
A
Decoder
A L U
OV

Runtime Library and SVCT Example

User Program

-

-

SVC(4)

-

-

-

-

Runtime Library for "Read"

SVCFLAG=1

LOADIP OLD-IP

I.H. searching code for "Read"

IF SVCFLAG=1

IP ← SCVT[B]

LOADIP OLD-IP

Address Open	Address Close	Address Write	Address Read	Address End
1	2	3	4	5

SCVT

The IP is overwritten!!!

User Program

-
-
SVC(4)
-
-
-
-

Runtime Library for "Read"

SVCFLAG=1

LOADIP OLD-IP

I.H. searching code for "Read"

IF SVCFLAG=1
IP ← SCVT[B]

LOADIP OLD-IP

When SVC(4) is executed "OLDIP ← IP" and after executing "SVCFLAG = 1", "OLDIP ← IP" in the interrupt cycle.

80 SVC(index)

80 SVC(index)

OLDIP ← IP;

B ← IR.ADDRESS

IP ← RTL-ADDRESS

DECODER ← 05

- Save IP of current program
- The Index value is temporarily loaded into register B
- Address of Runtime Library
- Transfer to Interrupt Cycle

05 Interrupt Cycle

If OV=1 Then IP← NEWIP; MODE ← 1 (ABEND)

If MP=1 Then IP← NEWIP; MODE ← 1 (ABEND)

If PI=1 Then IP← NEWIP; MODE ← 1 (ABEND)

IF I/O = 1 THEN OLDIP← IP;
IP ←NEWIP;
MODE←1;

If SVC=1, THEN OLDIP ←IP;
IP← NEWIP;
MODE ← 1;

DECODER ←00

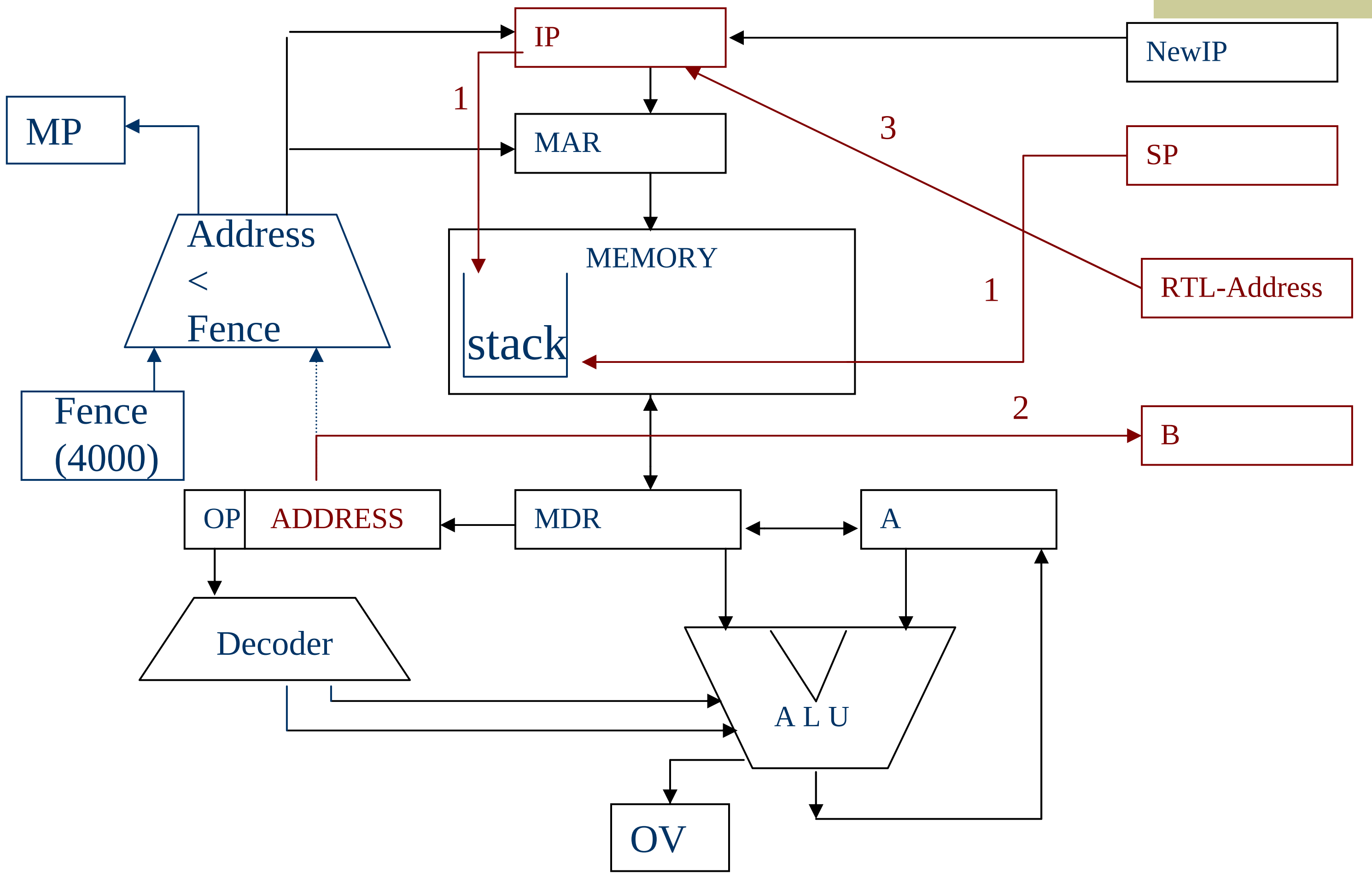
How can we handle nested interrupts?

Introducing the concept of a “Stack”.

1.- The “OLDIP” register is used as an stack pointer

2.- OLDIP register will be rename Stack Pointer (SP)

The Stack will store all return addresses



05 Interrupt Cycle
Including the stack mechanism

If OV=1 Then IP← NEWIP; MODE ← 1 (ABEND)

If MP=1 Then IP← NEWIP; MODE ← 1 (ABEND)

If PI=1 Then IP← NEWIP; MODE ← 1 (ABEND)

IF I/O = 1 THEN MEM[SP] ←IP; SP ← SP +1
IP ←NEWIP;
MODE←1;

If SVC=1, THEN MEM[SP] ←IP; SP ← SP +1
IP← NEWIP;
MODE ← 1;

DECODER ←00

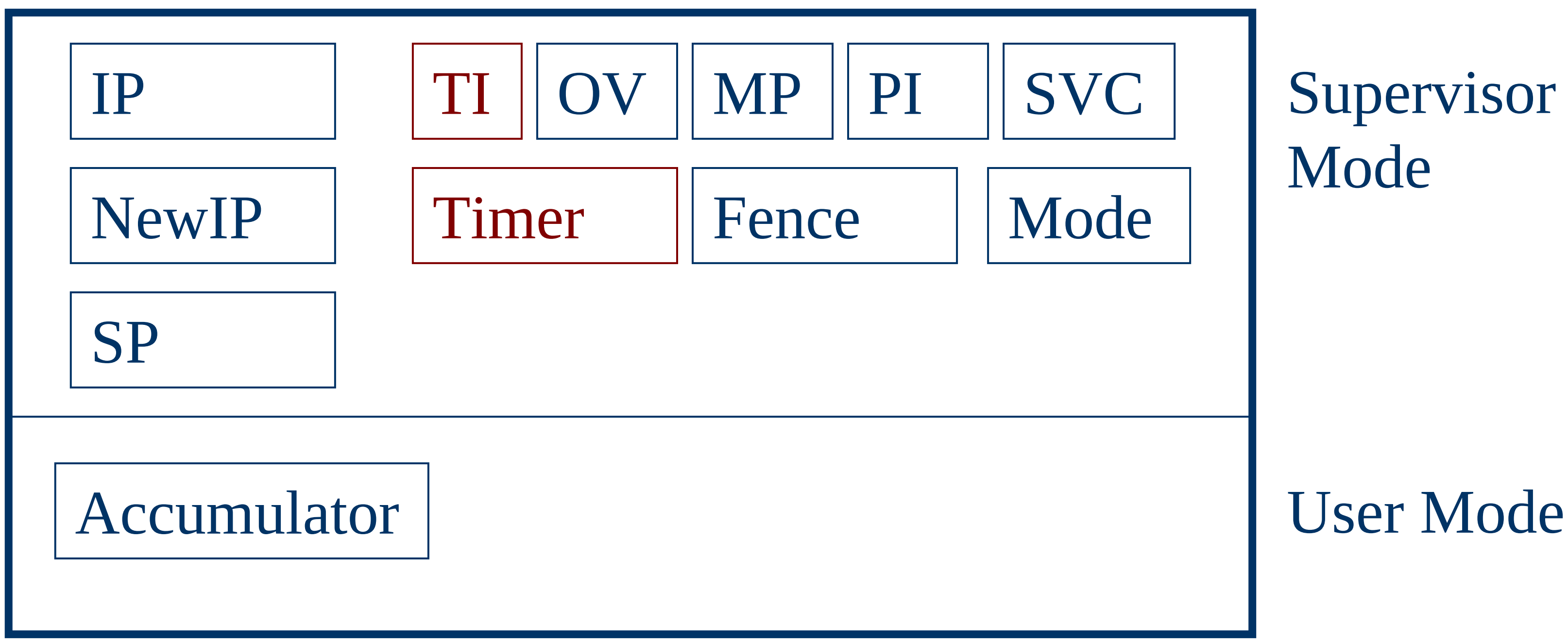
Program State Word including the SVC flag

IP	Interrupt Flags						MASK	Mode
	OV	MP	PI		I/O	SVC	To be defined later	

Timer Interrupt

- What if a program has an infinite loop?
- We can add a time register, set to a specific value before a program stops, which is decremented with each clock tick
- When the timer reaches zero, the Timer Interrupt bit (TI) is set to "1", indicating that a timer interrupt has occurred and transferring control to the interrupt handler
- Prevents a program from monopolizing the CPU

Timer Interrupt cont.



Program State Word

IP	Interrupt Flags						MASK	Mode
	OV	MP	PI	TI	I/O	SVC	To be defined later	

Interrupt Vector

- Switching between user and supervisor modes must be done as quickly as possible
- In the case of the VN machine, control is transferred to the interrupt handler, which then analyzes the flags and determines which is the appropriate course of action to take.
- A faster form of switching directly to the procedure or routine that handles the interrupt can be implemented using an *interrupt vector*

Interrupt Vector, cont.

- The idea of an interrupt vector consists of partitioning the interrupt handler into several programs, one for each type of interrupt.
- The starting addresses of each program are kept in an array, called the **interrupt vector**, which is stored in main memory.

Interrupt Vector Structure

- For each type of interrupt, there is a corresponding entry in the array, called IHV.
- Instead of transferring control just to the Interrupt Handler, we specify the element in the array that corresponds to the interrupt that occurred.
- This way, the routine that handles that interrupt is automatically executed.

05 Interrupt Cycle with the Interrupt Vector

If OV=1 Then IP ←IHV[0]; Mode ←1

If MP=1 Then IP ←IHV[1]; Mode ←1

If PI=1 Then IP ←IHV[2]; Mode ←1

If TI=1 Then MEM[SP] ←IP; SP ← SP +1;
IP ←IHV[3];
MODE ←1;

0	OV
1	IP
2	PI
3	TI
4	I/O
5	SVC

05 Interrupt Cycle with the Interrupt Vector, Cont.

If I/O=1 Then MEM[SP] ← IP; SP ← SP +1;
IP ← IHV[4];
MODE ← 1;

If SVC=1 Then MEM[SP] ← IP; SP ← SP +1;
IP ← IHV[5];
MODE ← 1;

DECODER ← 00;

0	OV
1	IP
2	PI
3	TI
4	I/O
5	SVC

Multiprogramming and Timers

- *Multiprogramming*: allowing two or more user programs to reside in memory
- If we want to run both programs, each program, P1 and P2, can be given alternating time on the CPU, letting neither one dominate CPU usage.

Process Concept

In order to implement multiprogramming we need to utilize the concept of a *process*.

Process: defined as a program in execution

We'll explore this concept further in the next lecture.