Lecture 19

COP3402 FALL 2015 – DR. MATTHEW GERBER – 11/25/2015

FROM EURIPIDES MONTAGNE, FALL 2014

Tonight

- System Calls
- The Vector Table
- Runtime Libraries
- Interrupt Vectors

The System Call Vector Table

We discussed *system calls* last session.

- System calls are actually programs – or at least subprograms – since they require multiple instructions to execute
- System calls are really executed by *runtime libraries* that contain instructions to execute the call
  - These libraries are typically written either in assembly language or in another language with very low-level capabilities, such as C

The *System Call Vector Table* (SCVT) contains the memory addresses for the beginning of each system call.

- Each of these addresses is a pointer into a runtime library

Runtime Libraries

Runtime libraries are precompiled procedures that can be called at runtime.

- Some runtime libraries are simply libraries of user-level code – they provide functionality like string manipulation, or interpretation of regular expressions
- Other runtime libraries are system-level libraries, or simply *system libraries*

We need system libraries to be able to execute privileged instructions

- We do this with yet another flip-flop
- We'll call this new flip-flop the SVC flag
- System libraries set this flag to 1
- This causes the system to switch to supervisor mode in the next interrupt cycle

System libraries…

- Are shared by all user programs
- *Cannot* be modified by *any* user program

The SVC Instruction

We create a new instruction SVC (index) for system calls. We'll assign it opcode 80.

The index (passed as IR.ADDR) is the entry point to the SCVT.

- So a compiler will eventually translate the read() library call into the SVC instruction
- …and pass an IR.ADDR corresponding to the system call index for the read function

We're notionally expanding the tiny machine here and assuming we have another register called B.

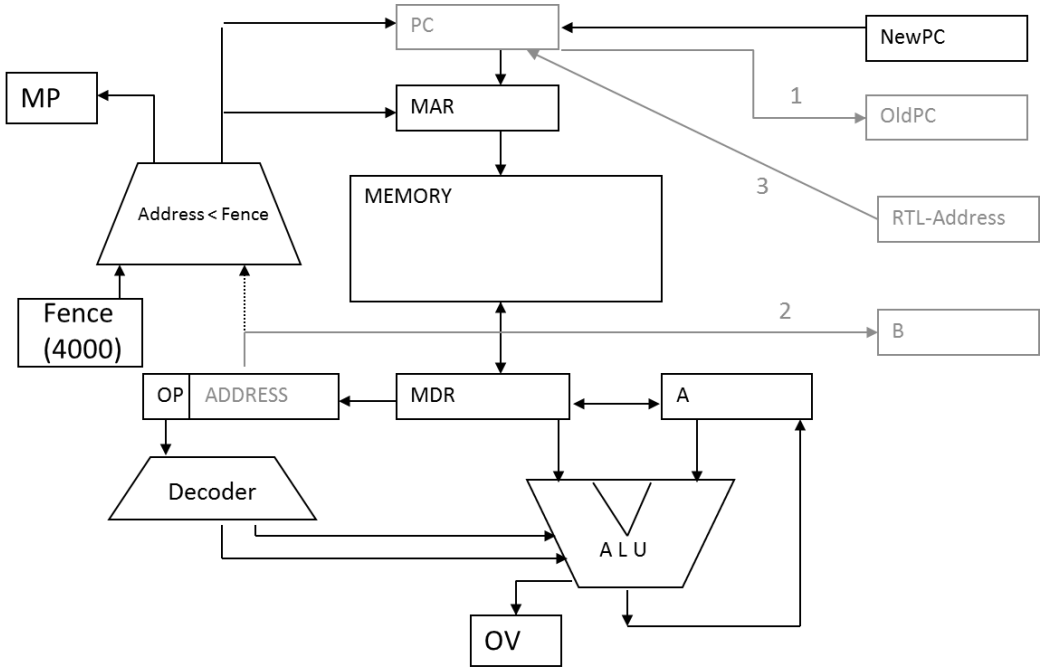
80 SVC(index)

OLDPC ← PC;	/* Save current PC */
B ← IR.ADDRESS	/* Which call? */
PC ← RTL-ADDRESS	/* Which library? */
DECODER ← 05	/* Supervisor */

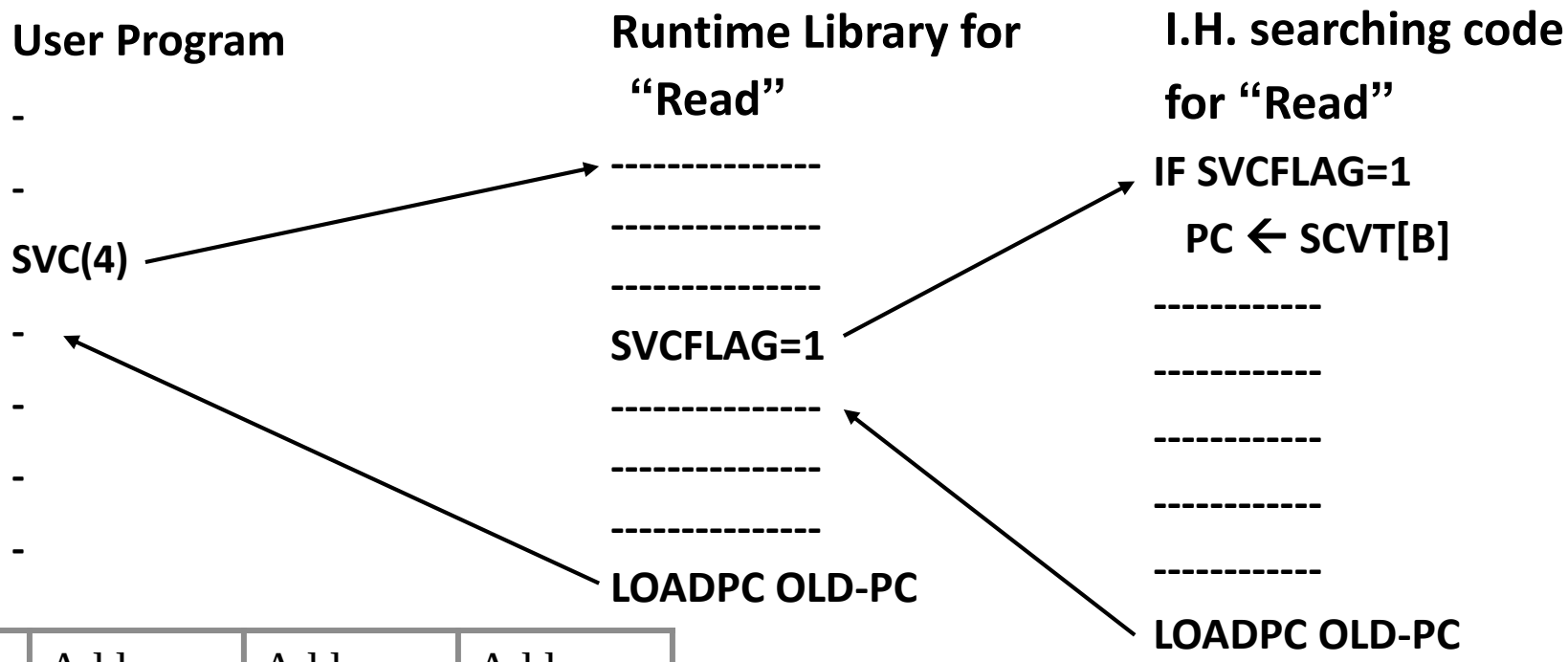
The Increasingly Less Tiny Machine

Yes, our little CPU is getting more and more complicated.

No, we're not done. In fact, we've got a problem.



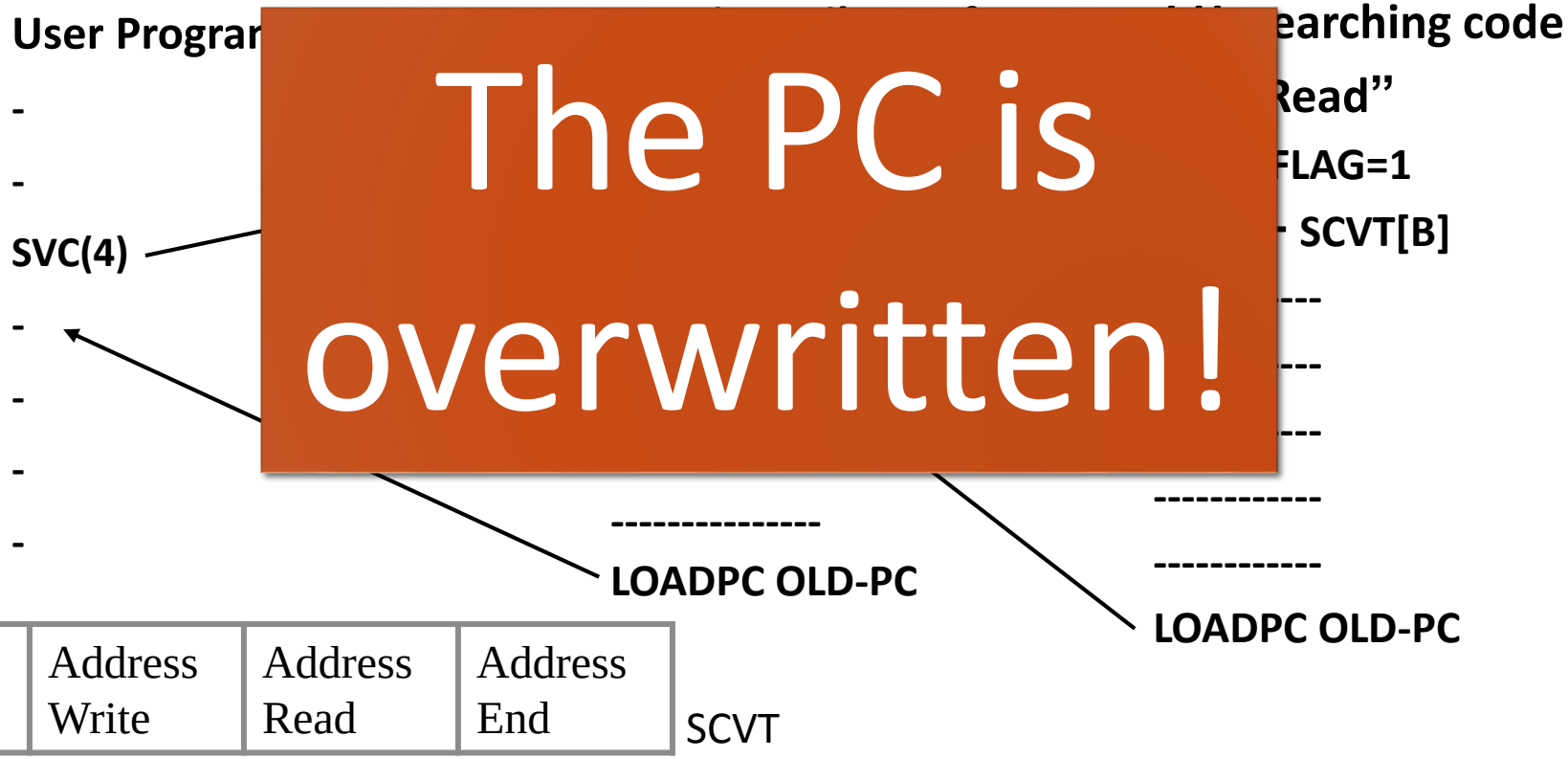
Runtime Library and SVCT Example



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

SCVT

Runtime Library and SVCT Example



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

SCVT

The Culprits

INSTRUCTION F

```
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 instr */
  PC ← NEWPC; MODE ← 1        /* ABEND */
ELSE IF I/O = 1 THEN          /* I/O interrupt */
  OLDPC ← PC
  /* Save process state here */
  PC ← NEWPC
  MODE ← 1
END IF
DECODER ← 00
```

THE SYSTEM CALL

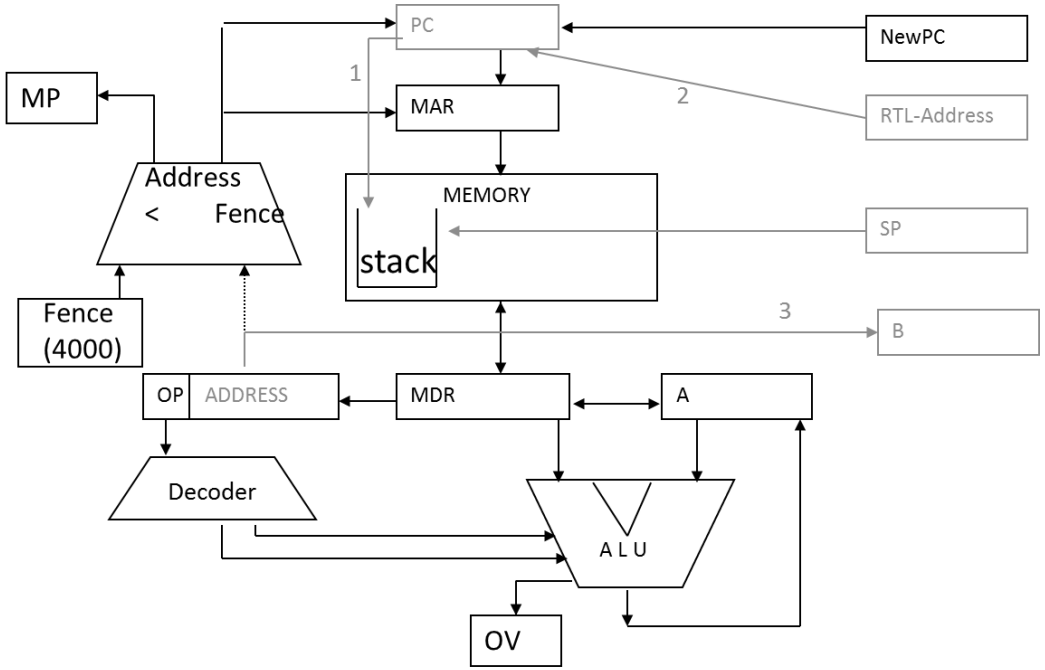
80 SVC(index)

```
OLDPC ← PC;          /* Save current PC */
B ← IR.ADDRESS       /* Which call? */
PC ← RTL-ADDRESS     /* Which library? */
DECODER ← 05         /* Supervisor */
```

How can we handle nested interrupts?

By turning OldPC into a stack.

- We use what was the OldPC register as a stack pointer.
- In fact, we'll rename it **SP**, for, well, Stack Pointer.



Instruction F with the Stack

Here's our supervisor call with the new stack model in place.

```
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 instr */
  PC ← NEWPC; MODE ← 1          /* ABEND */
ELSE IF I/O = 1 THEN            /* I/O interrupt */
  MEM[SP] ← PC
  SP ← SP +1
  /* Save process state here */
  PC ← NEWPC
  MODE ← 1
END IF
DECODER ← 00
```

PSW with SVC

And here's the program state word with the SVC flag.

Notice that there's one more interrupt flag to go? Let's look at that one now.

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

Timer Interrupts

How can we keep a program that has an infinite loop from monopolizing the CPU?

- We can add a **time register**, set to a specific value whenever a program is given control by the supervisor
- The register decrements with each clock tick
- When the timer reaches zero, the Timer Interrupt bit (TI) is set to “1”
- This indicates that a **timer interrupt** has occurred…
- …and, of course, control is transferred to the supervisor

PSW with TI and CC

Here's the program state word with that last interrupt flag – TI.

While we're at it, we add the ALU condition code bits – accumulator comparison results of Zero, Greater Than and Less Than.

PC	Interrupt Flags						MASK	CC			Mode
	OV	MP	PI	TI	I/O	SVC	To be defined later	G	Z	L	

The Interrupt Vector

We have to switch between user and supervisor modes quite a lot.

- Every system call requires it
- Even something as trivial as a file read

This switch must, therefore, be as fast as possible.

In the case of our current machine, control is transferred to the interrupt handler, which then analyzes the flags and determines the appropriate course of action.

We can speed this up with an *interrupt vector*.

- An array of addresses of components of the interrupt handler
- One subprogram for each type of interrupt
- Allows for simpler subprograms and avoids a level of comparison

The Interrupt Cycle with a Vector

Interrupt #	Address
0	OV
1	MP
2	PI
3	TI
4	I/O
5	SVC

```
IF OV = 1 THEN                  /* Overflow */
  PC ← IHV[0]; MODE ← 1         /* ABEND */
ELSE IF MP = 1 THEN             /* Memory violation */
  PC ← IHV[1]; MODE ← 1         /* ABEND */
ELSE IF PI = 1 THEN             /* Invalid instr */
  PC ← IHV[2]; MODE ← 1         /* ABEND */
ELSE IF TI = 1 THEN             /* Timer */
  MEM[SP] ← PC; SP ← SP +1
  PC ← IHV[3]; MODE ← 1
ELSE IF I/O = 1 THEN            /* I/O interrupt */
  MEM[SP] ← PC; SP ← SP +1
  PC ← IHV[4]; MODE ← 1
ELSE IF I/O = 1 THEN            /* System call */
  MEM[SP] ← PC; SP ← SP +1
  PC ← IHV[5]; MODE ← 1
END IF
DECODER ← 00
```

Next Time: Multiprogramming