

CS2200
Systems and Networks
Spring 2022

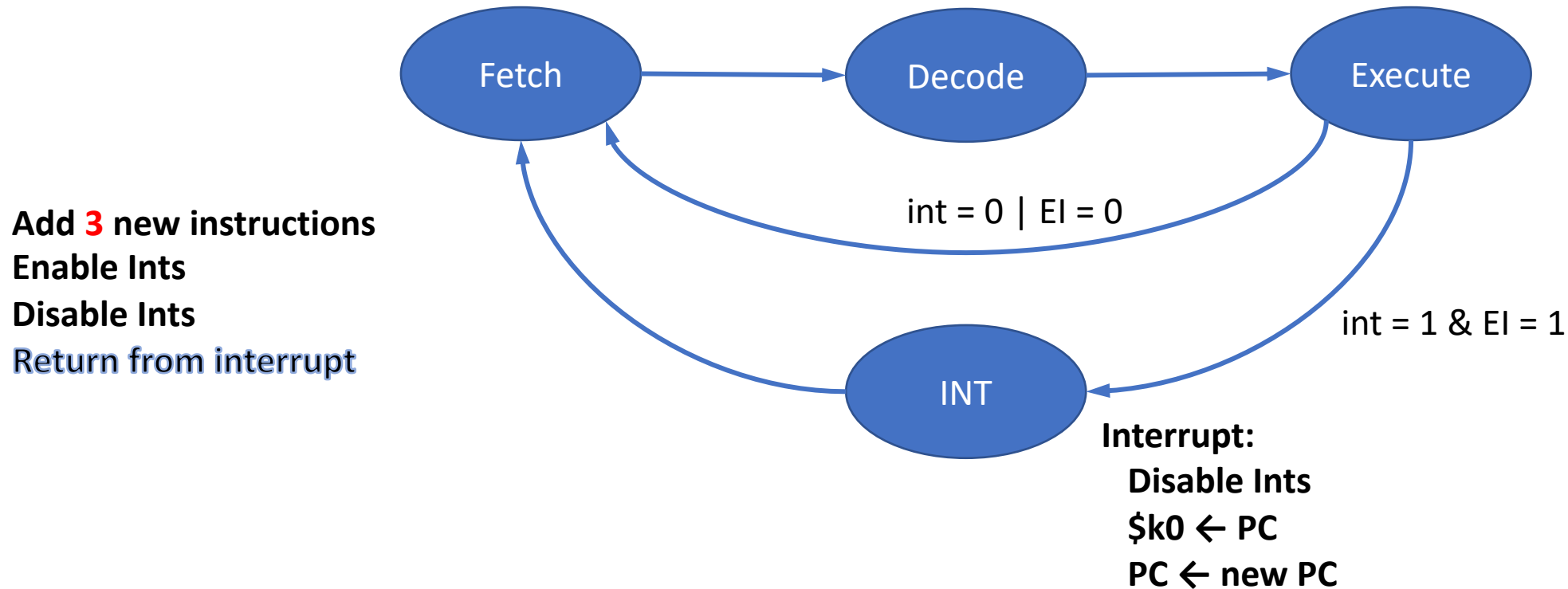
Lecture 8: Interrupts (cont'd)
and Performance

Alexandros (Alex) Daglis
School of Computer Science
Georgia Institute of Technology
adaglis@gatech.edu

Announcements

- Homework 2 is out
- Change in lab organization
 - Labs are mandatory but attendance is not a graded component
 - Every lab section is live streamed
 - One section is recorded
- Finishing Chapter 4, starting Chapter 5

Summary of provisions to handle interrupts



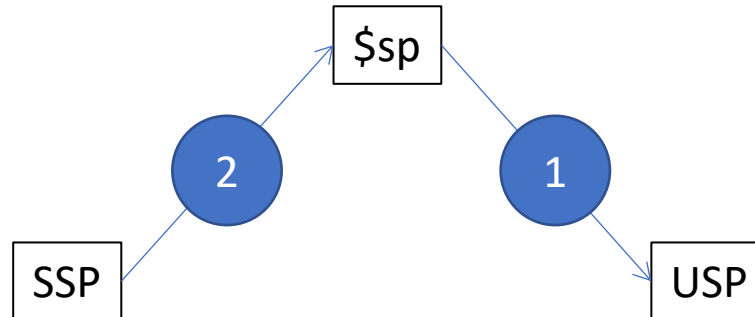
- Upon an interrupt, store the current PC implicitly into a special register \$k0, disable interrupts, and set the PC to the address of the handler
- Upon returning from an interrupt (RETI), store \$k0 into the PC and enable interrupts.

Where to save/restore CPU registers in the interrupt handler

- The user stack?
- Bad idea. The user doesn't even have to set \$sp if he doesn't feel like it. Bad practice, but real possibility.
- Where, then?
- How about we let the OS have a system stack that we know is handled properly?

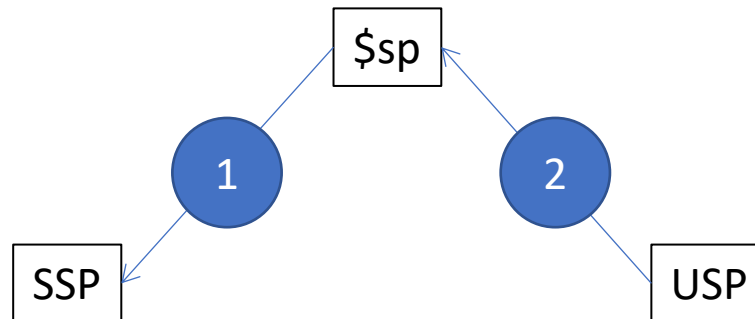
Stack for saving/restoring

- Hardware has no guarantee for stack behavior by user program (register/conventions)
- Equip processor with 2 saved stack pointers (User/System)
- On interrupt, save user stack pointer from \$sp and restore the system stack pointer to \$sp
- We'll need two more registers, USP and SSP

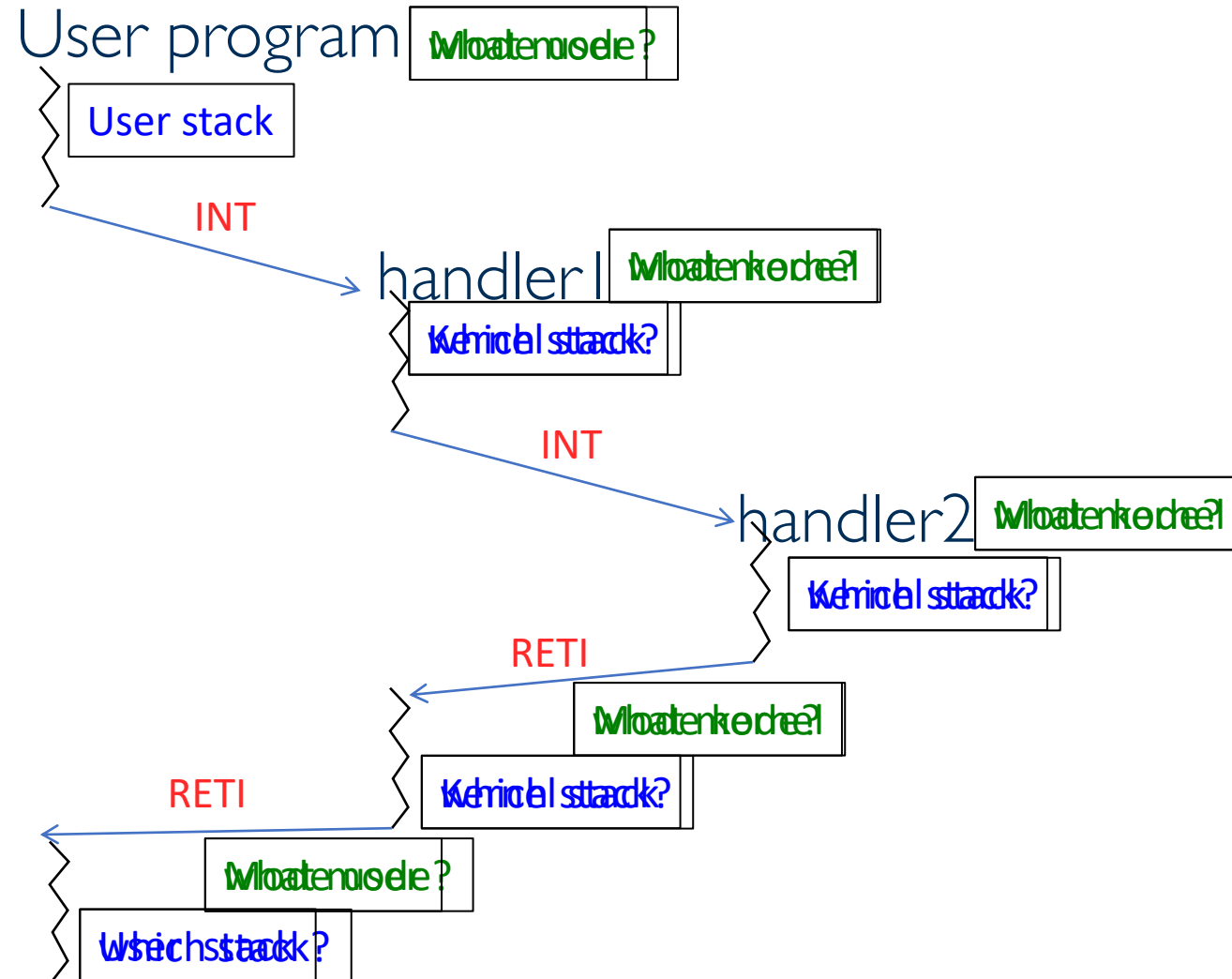


Stack for saving/restoring

- Use system stack for saving all necessary information
- Upon completion of interrupt restore registers, etc.
- The restore user stack pointer by reversing earlier swap
- Keep a user/kernel mode flag to record whether we're using the user or kernel stack



Stacks and modes during interrupts



Summary of interrupt actions

INT macro state:

$\$k0 \leftarrow PC$

Assert INTA to acknowledge interrupt

Receive IV (interrupt vector) from the device on the data bus

$PC \leftarrow \text{Mem}[\text{IV}]$

if user mode,

$USP \leftarrow \$sp; \$sp \leftarrow SSP$

Push mode on stack

$\text{mode} \leftarrow \text{kernel}$

Disable interrupts

RETI instruction:

$PC \leftarrow \$k0$

Pop mode from system stack

if user mode,

$SSP \leftarrow \$sp; \$sp \leftarrow USP$

Enable interrupts

A working interrupt handler

Handler:

```
// handler starts with interrupts disabled
push $k0 onto system stack;
enable interrupts;

save processor registers to system stack;
execute device code;

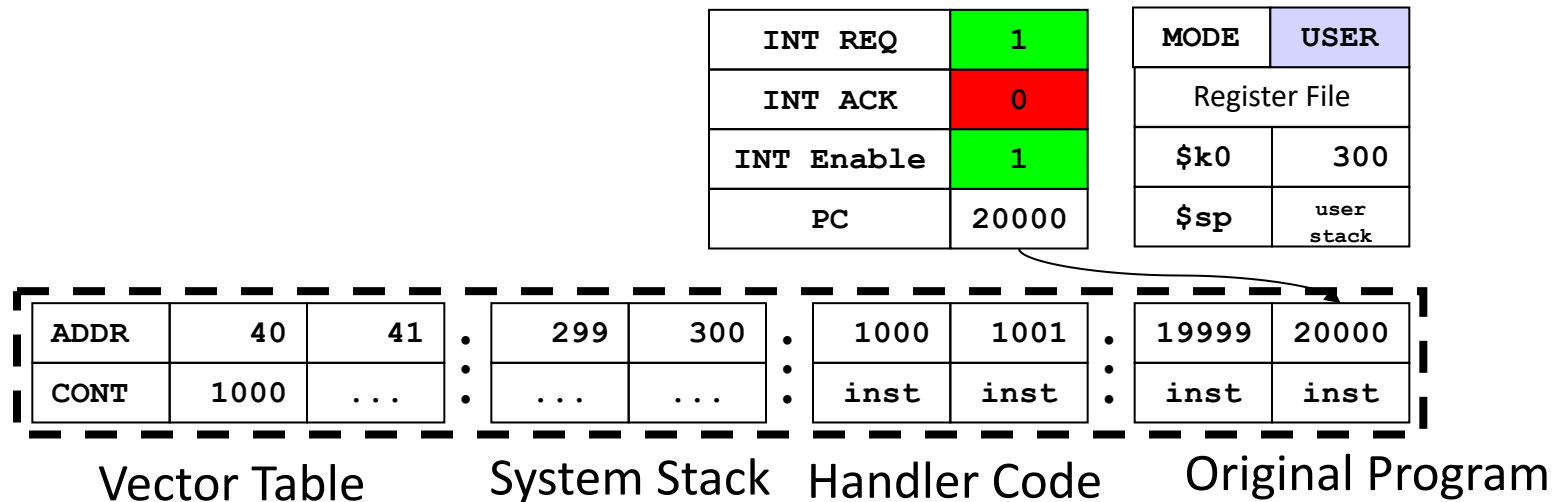
restore processor registers from system stack;
disable interrupts;
pop $k0 from system stack;
// handler ends with interrupts disabled
return to original program using RETI;
```

Architecture enhancements to LC-2200 for interrupts

1. An interrupt vector table (IVT), to be initialized by the operating system with handler addresses.
2. An exception/trap register (ETR) that contains the vector for internally generated exceptions and traps.
3. A Hardware mechanism for receiving the vector for an externally generated interrupt.
4. User/kernel mode and associated mode bit in the processor.
5. User/system stack corresponding to the mode bit.
6. A hardware mechanism for storing the current PC implicitly into a special register \$k0, upon an interrupt, and for retrieving the handler address from the IVT using the vector (either internally generated or received from the external device).
7. Three new instructions to LC-2200:
 - Enable interrupts
 - Disable interrupts
 - Return from interrupt

Putting it all together

Executing instruction at 19999. The PC has already been incremented. Device signals interrupt in middle of instruction. \$sp points to user stack



Putting it all together

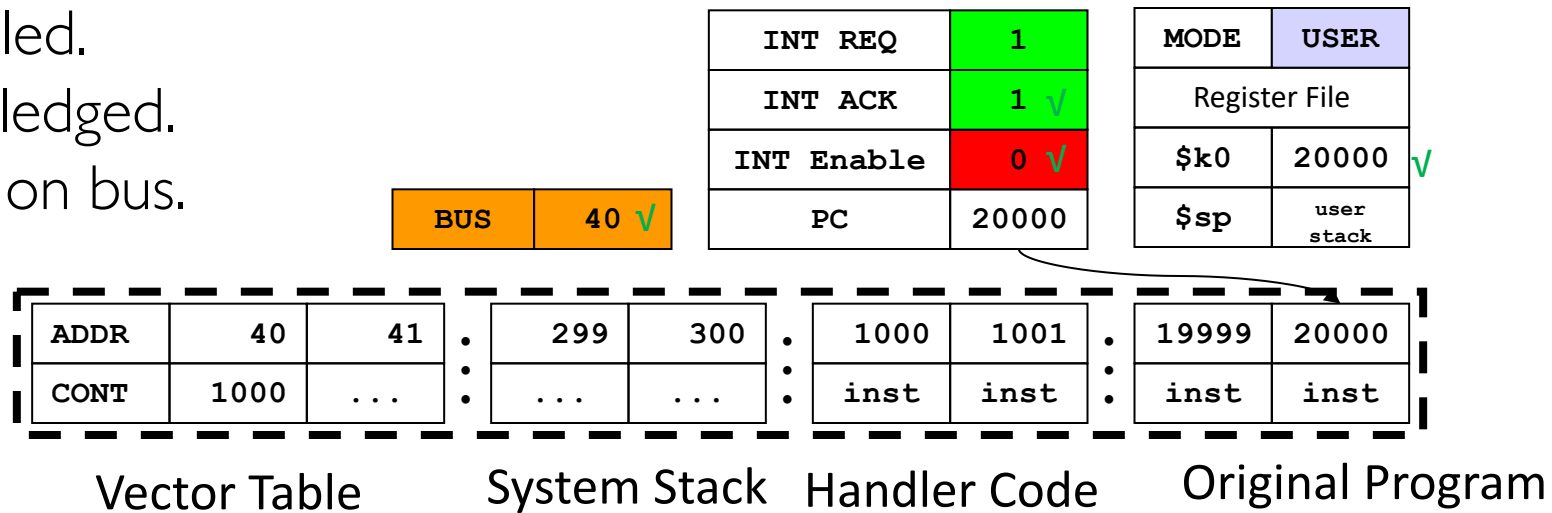
Interrupt has been noticed.

\$k0 gets PC.

Interrupts are disabled.

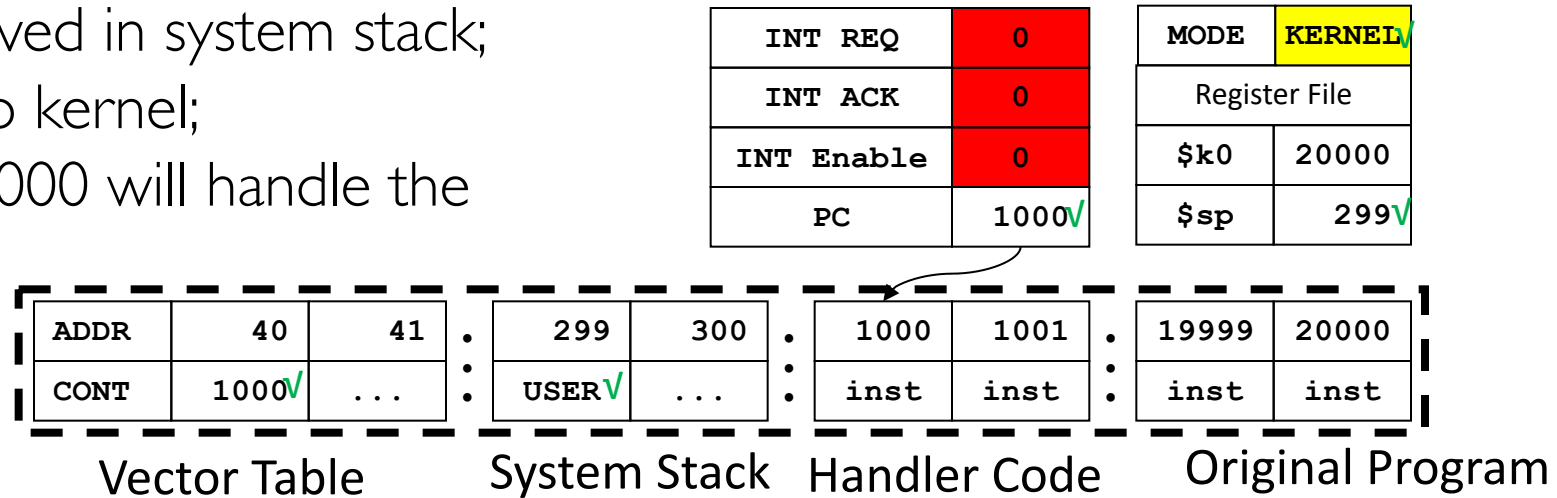
Interrupt is acknowledged.

Device puts vector on bus.



Putting it all together

Handler address is put into PC
\$sp now points to system stack;
Current mode is saved in system stack;
New mode is set to kernel;
Interrupt code at 1000 will handle the interrupt.



Putting it all together

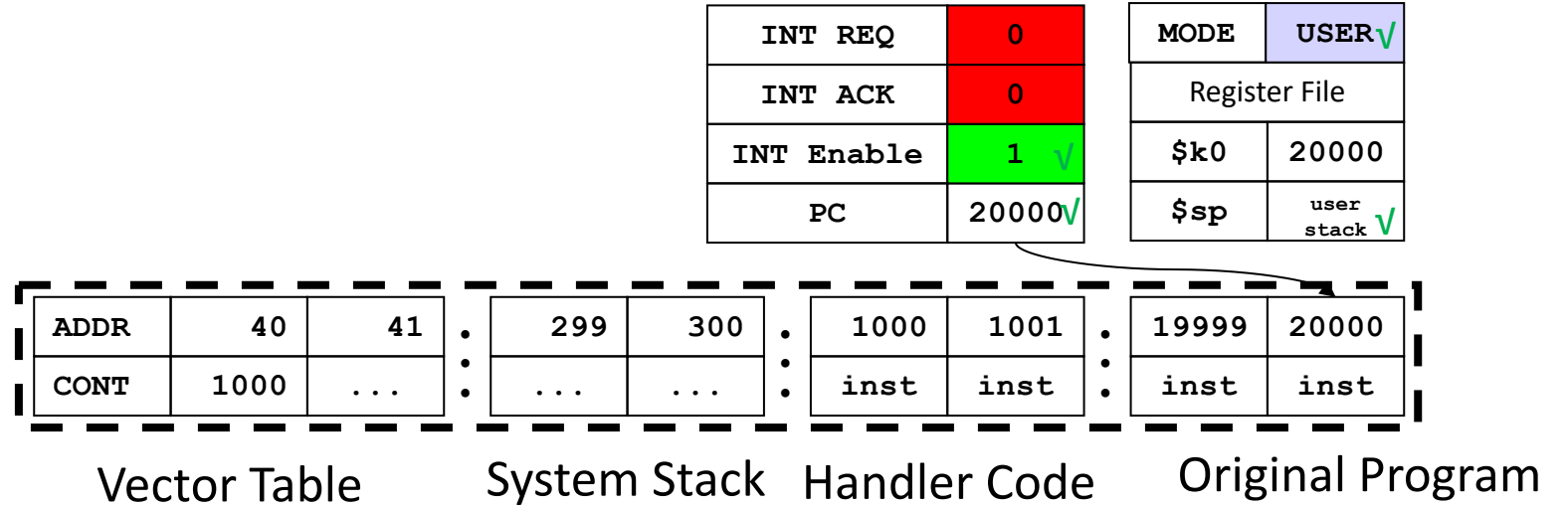
Handler completes.

RETI instruction restores mode from system stack; since returning to user program in this example, sets Mode to User;

\$sp now points to user stack;

copies \$k0 into PC;

re-enables interrupts



Summary

- Interrupts help a processor communicate with the outside world.
- An interrupt is a specific instance of program discontinuity.
- Processor/Bus enhancements included
 - Three new instructions
 - User stack and system stack pointers
 - Mode bit
 - INT macro state
 - Control lines called INT and INTA

Summary

- Software mechanism needed to handle interrupts; traps and exceptions are similar.
- Discussed how to write a generic interrupt handler that can handle nested interrupts.
- Intentionally simplified. Interrupt mechanisms in modern processors are considerably more complex. For example, modern processors categorize interrupts into two groups: *maskable* and *non-maskable*.
 - maskable: Interrupts that can be temporarily turned off
 - Non-maskable: Interrupts that cannot be turned off

Summary

- Presented mode as a characterization of the internal state of a processor. Intentionally simplistic view.
- Processor state may have a number of other attributes available as discrete bits of information (similar to the mode bit).
 - Modern processors aggregate all of these bits into one register called *processor status word (PSW)*.
 - Upon an interrupt and its return, the hardware implicitly pushes and pops, respectively, both the PC and the PSW on the system stack.
- The interested reader is referred to more advanced textbooks on computer architecture for details on how the interrupt architecture is implemented in modern processors.

Summary

- Presented simple treatment of the interrupt handler code to understand what needs to be done in the processor architecture to deal with interrupts. The handler would typically do a lot more than save processor registers.
- LC-2200 designates a register \$k0 for saving PC in the INT macro state. In modern processors, there is no need for this since the hardware automatically saves the PC on the system stack.

Performance

Metrics

If we're going to try to make processors better, we're going to have to take measurements

Common Metrics:

- Space → memory footprint
- Time → execution time
- Instruction frequency
 - Static
 - Dynamic
- Benchmarks

What determines execution time?

- CPI = "Cycles Per Instruction" → number of clock cycles each instruction takes
- *Execution time* = $(\sum CPI_j) * \text{clock cycle time}$,
where $1 \leq j \leq n$
- That's a pretty tough sum to compute because modern computers can execute billions of instructions per second
- So, we approximate as
Execution time = $n * CPI_{Avg} * \text{clock cycle time}$,
where n is the number of instructions (executed--not static--instruction count)
- This is known as the **Iron Law** of processor performance

The “Iron Law” of Processor Performance

$$\text{Processor Performance} = \frac{\text{Time}}{\text{Program}}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

ISA & Compiler

Microarchitecture

Circuit



What's the execution time?

1 GHz processor, 10K instructions, $CPI_{Avg} = 3$

- A. 3 msec
- B. 1 GHz
- C. 0.003 msec
- D. 30 μ sec
- E. 30K
- F. About 11, sir

$$10^{-9} \text{ sec/cy} \times 10^4 \text{ inst} \times 3 \text{ cy/inst}$$

$$= 3 \times 10^{-5} \text{ sec}$$

$$= .00003 \text{ sec}$$

$$= .03 \text{ msec}$$

$$= 30 \mu\text{sec}$$

Instruction Frequency

- *Static* instruction frequency refers to number of times a particular instruction occurs in compiled code.
 - Impacts memory footprint
 - If a particular instruction appears a lot in a program, can try to optimize amount of space it occupies by clever instruction encoding techniques in the instruction format.
- *Dynamic* instruction frequency refers to number of times a particular instruction is executed when program is run.
 - Impacts execution time of program
 - If dynamic frequency of an instruction is high then can try to make enhancements to datapath and control to ensure that CPI taken for its execution is minimized.



Static instruction frequency...

- A. Refers to the type of instructions in the instruction-set
- B. Refers to the frequency of occurrence of instructions in compiled code
- C. Refers to the frequency of occurrence of instructions that actually get executed ← **Dynamic instruction frequency**
- D. Refers to the clock frequency of the processor
- E. Is the basis for datapath design

What is the static frequency of ADD?

The ADD instruction occurs twice in a program that contains a total of 1000 instructions in the compiled code. All 1000 instructions get executed during a program run. One of the ADD instructions is in a 5-instruction loop that gets executed 1000 times.

$$2/1000 = 0.2\%$$



What about the **dynamic** frequency of ADD?

The ADD instruction occurs twice in a program that contains a total of 1000 instructions in the compiled code. All 1000 instructions get executed during a program run. One of the ADD instructions is in a 5-instruction loop that gets executed 1000 times.

- A. Two
- B. 0.2%
- C. $(1000/5995) * 100\%$
- D. $(1001/5995) * 100\%$
- E. $(1/5000) * 100\%$
- F. $(1001/5000) * 100\%$

ADDs executed:

$$1 + 1000 = 1001$$

Total executed:

$$(1000-5) \text{ inst} + 1000 * 5 = 5995$$

Dynamic add frequency: $1001/5995$

The need for speed!

- How do we improve performance?
- Reduce execution time (of course)
- How?
- The Iron Law tells us:

$$\text{Execution time} = N \downarrow * \text{CPI} \downarrow * \text{Cycle Time} \downarrow$$

- How can we reduce the left-hand side?
- Reduce one or more of the right-hand factors

How can we measure improvement?

$$\text{speedup}_{\text{A over B}} = \frac{\text{Execution Time On Processor B}}{\text{Execution Time On Processor A}}$$

$$\text{speedup}_{\text{improved}} = \frac{\text{Execution Time Before Improvement}}{\text{Execution Time After Improvement}}$$

$$\text{improvement in execution time} = \frac{\text{old execution time} - \text{new execution time}}{\text{old execution time}}$$

Example

You improve your application's algorithm so that it runs in 29 seconds instead of 38 seconds.

What is the speedup?

$$E_{\text{Before}}/E_{\text{After}}$$

$$38 / 29 = 1.3103 \\ = 31\% \text{ speedup}$$

$$29 * 1.31 \sim 38$$

What is the improvement in execution time?

$$(E_{\text{old}} - E_{\text{new}})/E_{\text{old}}$$

$$(38 - 29) / 38 = .2368... \\ = 24\%$$

$$38 - (24\% \text{ of } 38) \sim 29$$



Improvement in execution speed

It takes me 8 minutes to walk to class from my office. I can run twice as fast as I walk. If I walk half the distance and run the remaining half, how much time will I take to reach class from my office?

Rank

Responses

$$\begin{aligned}
 8 &= d/v \\
 n &= \frac{1}{2} d/v + \frac{1}{2} d/2v \\
 8v &= d \\
 n &= 4v/v + 4v/2v \\
 n &= 4 + 2 \\
 n &= 6
 \end{aligned}$$

Amdahl's Law

Amdahl's Law

$$\text{Time}_{\text{after}} = \frac{\text{Time}_{\text{affected}}}{\text{Amount of Improvement}} + \text{Time}_{\text{unaffected}}$$

My office walk: $6 = 4 / 2 + 4$

Improving an instruction

A processor spends 20% of its time on ADD instructions. An engineer proposes to improve the speed of the ADD instruction by 4 times. What is the speedup achieved by this modification?

The improvement only applies for the ADD instruction, so 80% of the execution time is unaffected by the improvement.

Original normalized execution time = 1

$$\begin{aligned}\text{New execution time} &= (\text{time spent in ADD}/4) + \text{remaining time} \\ &= 0.2 / 4 + .8 \\ &= 0.85\end{aligned}$$

$$\begin{aligned}\text{Speedup} &= \text{execution time before} / \text{execution time after} \\ &= 1 / 0.85 = 1.18 = 18\%\end{aligned}$$



Improving an instruction

An engineer is asked to improve the processor's overall performance by 2 times by optimizing the ADD instruction. The processor spends 20% of its time on ADD instructions. How much faster must the ADD instruction become?

- A. 2x
- B. 10x
- C. 100x
- D. That's impossible!!

Execution time?

CPI of Instruction Classes	Code 1	Code 2
R-type = 2	3	10
I-type = 10	3	1
J-type = 3	5	2
S-type = 4	2	3

13 inst
6+30
+15+8 =
59 cycles

16 inst
20+10+6
+12 =
48 cycles

- A. Code 1 is better than Code 2 since it has fewer instructions
- B. Code 2 is better than Code 1 since it has fewer instructions
- C. Code 1 is better than Code 2 since it takes fewer total clock cycles to execute
- D. Code 2 is better than Code 1 since it takes fewer total clock cycles to execute

Architecture change?

- We have a computer with three types of instructions that have the following CPI:

Type	CPI
A	2
B	5
C	1

- An architect determines that she can reduce the CPI for B to 3 but will need to slow the clock speed of the processor. What is the maximum permissible slowing of the clock that will make this change worthwhile?
- Assume that all the workloads for this processor use 30% of A, 10% of B, and 60% of C types of instructions

How do we answer that?

Execution time of the old machine:

$$ET_o = N * (F_{Ao} * CPI_{Ao} + F_B * CPI_{Bo} + F_C * CPI_{Co}) * C_o$$

(where F_x and CPI_x are the dynamic frequencies and CPIs of each type of instruction, respectively)

$$ET_o = N * (0.3 * 2 + 0.1 * 5 + 0.6 * 1) * C_o = N * 1.7 * C_o$$

Execution time for the new machine:

$$ET_n = N * (0.3 * 2 + 0.1 * 3 + 0.6 * 1) * C_n = N * 1.5 * C_n$$

For the design to be viable,

$$ET_n < ET_o$$

$$N * 1.5 * C_n < N * 1.7 * C_o$$

$$C_n < 1.7/1.5 * C_o$$

$$C_n < 1.13 * C_o$$

Maximum permissible increase in clock cycle time = 13%

Type	CPI
A	2
B	5
C	1

30% of A,
10% of B,
60% of C

Combining two instructions?

Instruction	CPI
Add	2
Shift	3
Others	2
Add/Shift	4

If the sequence ADD followed by SHIFT appears in 20% of the dynamic frequency of a program, what is the speedup of the program with all {ADD, SHIFT} replaced by the combined instruction?

[HINT: For every 10 instructions in the original program, 2 instructions are the ADD/SHIFT combo. Thus the number of instructions in the new program shrinks to 90% of the original program.]

A solution

$$\begin{aligned} ET_o &= N * (F_{ADD} * 2 + F_{SHIFT} * 3 + F_{others} * 2) * C \\ &= N * (0.1 * 2 + 0.1 * 3 + 0.8 * 2) * C \\ &= 2.1 * N * C \end{aligned}$$

With the combo instruction replacing {ADD SHIFT}, the number of instructions in the new program shrinks to 0.9N in the new program. The frequency of the combo instruction is 1/9 and the other instructions are 8/9.

$$\begin{aligned} ET_n &= (0.9 * N) * (F_{COMBO} * 4 + F_{others} * 2) * C \\ &= (0.9 * N) * (1/9 * 4 + 8/9 * 2) * C \\ &= 2 * N * C \end{aligned}$$

$$\begin{aligned} \text{Speedup} &= \text{old execution time} / \text{new execution time} \\ &= 2.1NC / 2NC \\ &= 1.05 \end{aligned}$$

Instruction	CPI
Add	2
Shift	3
Others	2
Add/Shift	4

Benchmarks

- **Benchmarks** are a set of programs that are **representative** of the workload for a processor.
- The key difficulty is to be sure that the benchmark program selected **really** are representative of the prospective workload.
- Standard benchmark suites (SPEC, LINPACK, Whetstone, Dhrystone, and many more) are used to try to compare “apples” with “apples” by summarizing performance *across a set of programs*
 - E.g., SPEC uses perl, gcc, AI apps, compression, imaging apps, modeling apps, etc. to represent a common workload
- A radical new design is hard to benchmark because there may not yet be a compiler or much code.

Using a benchmark

Some caveats:

- Composite functions come with caveats → we have to be very cautious using a single composite metric
- Testing a single system component (e.g., the processor) only gives a limited view: e.g., memory organization and memory—processor—bus bandwidths are also key
- Some processors do well on certain benchmark programs and other do well on other programs
- A composite index can be useful when we want to compare two processors without knowing the exact kind of workload they are going to run, but we must be very cautious
 - More on this later

Reasons to be skeptical of a benchmark

- The vendor gave you the benchmark results (in polite company, we call this a conflict of interest)
- The vendor wrote the benchmark suite
- The benchmark suite doesn't seem to have any elements that represent your workload (e.g., you run web server farms and the benchmark represents only computationally intensive scientific calculations)
- The equipment being benchmarked is different from the equipment you want to evaluate (maybe a little different, maybe a lot different)
- The benchmark uses a different compiler suite than you plan to use
- The cost of mistakenly choosing the wrong equipment is very high



Comparing Multiple Programs

	Computer A	Computer B	Computer C
Program 1 (secs)	1	10	20
Program 2 (secs)	1000	100	20
Program 3 (secs)	1001	110	40

A is 10 times faster than B for program 1
B is 10 times faster than A for program 2
A is 20 times faster than C for program 1
C is 50 times faster than A for program 2
B is 2 times faster than C for program 1
C is 5 times faster than B for program 2

Each statement above is correct...

...but I just want to know which machine is the best?

Need a composite metric



Let's Try a Simpler Example

Two machines timed on two benchmarks

	<u>Machine A</u>	<u>Machine B</u>
Program 1	2 seconds	4 seconds
Program 2	12 seconds	8 seconds

How much faster is Machine A than Machine B?

Attempt 1: ratio of runtimes, normalized to Machine A runtimes

program1: $4/2$

program2 : $8/12$

- Machine A ran 2 times faster on program 1, $2/3$ times faster on program 2
- On average, Machine A is $(2 + 2/3) / 2 = 4/3$ times faster than Machine B

It turns this “averaging” stuff can fool us; watch...

Example (con't)

Two machines timed on two benchmarks

	<u>Machine A</u>	<u>Machine B</u>
Program 1	2 seconds	4 seconds
Program 2	12 seconds	8 seconds

How much faster is Machine A than B?

Attempt 2: ratio of runtimes, normalized to Machine B runtimes

program 1: $2/4$ program 2 : $12/8$

- Machine A ran program 1 in $1/2$ the time and program 2 in $3/2$ the time
- On average, $(1/2 + 3/2) / 2 = 1$
- Put another way, Machine A is 1.0 times faster than Machine B

Example (con't)

Two machines timed on two benchmarks

	Machine A	Machine B
Program 1	2 seconds	4 seconds
Program 2	12 seconds	8 seconds

How much faster is Machine A than B?

Attempt 3: ratio of aggregated runtimes, norm. to A

- Machine A took 14 seconds for both programs
- Machine B took 12 seconds for both programs
- Therefore, Machine A takes $14/12$ of the time of Machine B
- Put another way, Machine A is $6/7$ faster than Machine B

Which is Right?

Question:

- How can we get three different answers?

Answer:

- Because, while they are all reasonable calculations...

...each answers a different question

Need to be more precise in understanding and posing these performance & metric questions

Arithmetic and Harmonic Mean

Average of the execution time that tracks total execution time is the arithmetic mean

$$\frac{1}{n} \sum_{i=1}^n \textit{Time}_i$$

This is the definition for “average” you are most familiar with

If performance is expressed as a rate, then the average that tracks total execution time is the harmonic mean

$$\frac{n}{\sum_{i=1}^n \frac{1}{\textit{Rate}_i}}$$

This is a different definition for “average” you are probably less familiar with

Geometric Mean

- Used for relative rate (i.e., ratio) or normalized performance

$$\textit{Relative_Rate} = \frac{\textit{Rate}}{\textit{Rate}_{ref}} = \frac{\textit{Time}_{ref}}{\textit{Time}}$$

- Geometric mean

$$\sqrt[n]{\prod_{i=1}^n \textit{Relative_Rate}_i} = \frac{\sqrt[n]{\prod_{i=1}^n \textit{Rate}_i}}{\textit{Rate}_{ref}}$$

Why does the choice of the mean matter?

Benchmark	Ops (millions)	Computer 1	Computer 2	Speedup (C2 vs C1)
<i>Absolute performance (Time)</i>				
Program 1	100	1	20	
Program 2	100	1000	20	
Total time		1001	40	25
Avg (arith mean)		500	20	25

Why does the choice of the mean matter?

Benchmark	Ops (millions)	Computer 1	Computer 2	Speedup (C2 vs C1)
<i>Absolute performance (Time)</i>				
Program 1	100	1	20	
Program 2	100	1000	20	
Total time		1001	40	25
Avg (arith mean)		500	20	25
<i>Performance in MFLOPS (Rate)</i>				
Program 1		100	5	
Program 2		0.1	5	
Arith. mean		50.1	5	0.1
Geom. mean		3.2	5	1.6
Harm. mean		0.2	5	25



Quiz

A car drives 30 mph for first 10 miles, 90 mph for next 10 miles

What is the car's average speed?

$$\text{Average speed} = (30 + 90)/2 = 60 \text{ mph}$$

- A. True
- B. False

Quiz

- E.g., 30 mph for first 10 miles, 90 mph for next 10 miles

- What is the average speed?

Average speed = $(30 + 90)/2 = 60$ mph (Wrong!)

- Correct answer:

Average speed = total distance / total time
 $= 20 / (10/30 + 10/90)$
 $= 45$ mph

For rates use Harmonic Mean!

Problems with Arithmetic Mean

- Applications do not have the same probability of being run
- Longer programs weigh more heavily in the average

For example, two machines timed on two benchmarks

	<u>Machine A</u>	<u>Machine B</u>
Program 1	2 seconds (20%)	4 seconds (20%)
<u>Program 2</u>	<u>12 seconds (80%)</u>	<u>8 seconds (80%)</u>

- If we do arithmetic mean, Program 2 “counts more” than Program 1
An X% improvement in Program 2 changes the average more than an X% improvement in Program 1
- But perhaps Program 2 is 4 times more likely to run than Program 1

Weighted Execution Time

Often, one runs some programs more often than others. Therefore, we should *weight* the more frequently used programs' execution time

$$\sum_{i=1}^n \text{Weight}_i \times \text{Time}_i$$

Weighted Harmonic Mean

$$\frac{1}{\sum_{i=1}^n \frac{\text{Weight}_i}{\text{Rate}_i}}$$

Using a Weighted Sum (or weighted average)

	<u>Machine A</u>	<u>Machine B</u>
Program 1	2 seconds (20%)	4 seconds (20%)
Program 2	12 seconds (80%)	8 seconds (80%)
Total	10 seconds	7.2 seconds

Allows us to determine relative performance $10/7.2 = 1.39$

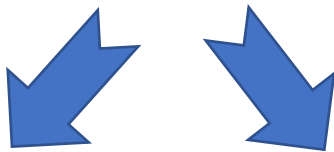
→ Machine B is 1.39 times faster than Machine A

(instead of $14/12 = 1.16x$ faster without weighting)

What if we only know normalized runtimes?

Normalize runtime of each program to a reference

	<u>Machine A (ref)</u>	<u>Machine B</u>
Program 1	2 seconds	4 seconds
Program 2	12 seconds	8 seconds



	<u>Machine A (norm to B)</u>	<u>Machine B (norm to A)</u>
Program 1	0.5	2.0
Program 2	1.5	0.666
Average?	1.0	1.333

- When we normalize A to B and average, it looks like A & B are the same.
- But when we normalize B to A and average, it looks like A is better!

Using Geometric Mean

	Machine A (norm to B)	Machine B (norm to A)
Program 1	0.5	2.0
Program 2	1.5	0.666
Geometric Mean	0.866	1.154

Note that $1.154 = 1/0.866$

Drawbacks:

- Does not reflect actual runtime because it normalizes
- Each application now counts equally

When is geomean useful?

Geometric mean of ratios is not proportional to total time

Use to compare machines when

- Relative performance on each program is known
- Relative runtime/weights of different programs is not known
- E.g., to aggregate speedups on set of programs

Rule of thumb: Use AM for times, HM for rates, GM for ratios

Summary of metrics

Name	Notation	Units	Comment
Memory footprint	-	Bytes	Total space occupied by the program in memory
Execution time	$(\sum \text{CPI}_j) * \text{clock cycle time, where } 1 \leq j \leq n$	Seconds	Running time of the program that executes n instructions
Arithmetic mean	$(E_1 + E_2 + \dots + E_p)/p$	Seconds	Average of execution times of constituent p benchmark programs
Weighted Arithmetic mean	$(f_1 * E_1 + f_2 * E_2 + \dots + f_p * E_p)$	Seconds	Weighted average of execution times of constituent p benchmark programs
Geometric mean	$p^{\text{th}} \text{ root } (E_1 * E_2 * \dots * E_p)$	Seconds	p^{th} root of the product of execution times of p programs that constitute the benchmark
Static instruction frequency		%	Occurrence of instruction i in compiled code
Dynamic instruction frequency		%	Occurrence of instruction i in executed code
Speedup (M_A over M_B)	E_B/E_A	Number	Speedup of Machine A over B
Speedup (improvement)	$E_{\text{Before}}/E_{\text{After}}$	Number	Speedup After improvement
Improvement in Exec time	$(E_{\text{old}} - E_{\text{new}})/E_{\text{old}}$	Number	New Vs. old
Amdahl's law	$\text{Time}_{\text{after}} = \text{Time}_{\text{unaffected}} + \text{Time}_{\text{affected}}/x$	Seconds	x is amount of improvement