

CS 2200 Exam 2 Notes

Krish Katariya

Last updated: **October 02, 2024**

Contents

1. Interrupts, Exceptions, and Traps	1
1.1. Synchronous vs Asynchronous	1
1.2. Interrupts	1
1.3. Exceptions	1
1.4. Traps	1
1.5. Dealing with Program Discontinuities	2
1.6. Modifications to FSM (INT)	2
1.6.1. INT Macro State	2
1.6.2. Cascaded Interrupts	3
1.7. RTI	3
1.8. Datapath Details for Interrupts	3
1.9. Stack for saving/restoring	4
2. Processor Speed Metrics	4
2.1. Benchmarks	4
3. Pipelining	5

1. Interrupts, Exceptions, and Traps

1.1. Synchronous vs Asynchronous

Definition 1.1.1

Synchronous events are events that are part of the intended events that are supposed to happen.

Asynchronous events are events that are unexpected or not part of the plan.

1.2. Interrupts

Definition 1.2.1

An **interrupt** is the mechanism by which devices catch the attention of the processor. This is an unplanned discontinuity from the unplanned program and is therefore **asynchronous**.

1.3. Exceptions

Definition 1.3.1

An **exception** is when the program unintentionally performs certain illegal operations such as divide by zero or follow an executing path unintended in the program specification.

1.4. Traps

Definition 1.4.1

A **trap** is when people make *system calls* to read/write files or for other such services from the system.

In this case, the program is accessing parts of the system whose integrity affects a whole community of users, which is why we need to do them through traps.

This allows the program to *fall into* the operating system, which will then decide what the user program wants and executes it.

1.5. Dealing with Program Discontinuities

Most of what the processor has to do to deal with program discontinuity is the same regardless of the type. A **handler** is the procedure that is executed when a discontinuity occurs, and the code for the handler is very similar to writing other procedures.

Definition 1.5.1

The **interrupt vector table** (IVT) stores a unique number (*vector*) which serves as a unique index into the IVT to get the correct handler.

In the case of traps and exceptions, the hardware automatically generates this vector internally, using the **exception/trap register** (ETR). For example, when the divide by zero error happens, the unique number associated with divide by zero errors will be placed in the ETR.

The partnership between OS and hardware for dealing with program discontinuities:

1. The architecture may itself define a set of exceptions and specify the numbers (vector values) associated with them. These are usually due to runtime errors encountered by program execution.
2. The operating system may define its own set of traps (systems calls) and the associated numbers
3. The operating system sets up the IVT at boot time with the address of handlers for dealing with program discontinuities
4. The hardware detects external interrupts and receives the vector value corresponding to the interrupting devices
5. The hardware uses the vector value as an index in IVT to retrieve the handler address and transfer control of the currently executing program

1.6. Modifications to FSM (INT)

The FSM now checks at the point of completion of an instruction if there is a pending interrupt. If there is (INT=y), then the FSM transitions to the INT macro state, and if not it resumes to Fetch.

1.6.1. INT Macro State

Theorem 1.6.1.1

1. First, we have to save the current PC value somewhere. We reserve the processor register **k0** (general purpose register 12) for this purpose. The INT macrostate saves PC into **k0**
2. We receive the PC value of the handler address from the device, load it into the PC and go to FETCH.

Theorem 1.6.1.2

Interrupt handler job:

- Enable interrupts
- Save processor registers
- Disable interrupt

1.6.2. Cascaded Interrupts

If there is an interrupt during an interrupt, the value saved in **k0** will get overwritten and it will lose track of the original PC value. This are called **cascaded interrupts**.

To fix cascaded interrupts we modify our INT / FSM implementation a little bit.

We add two instructions: **disable interrupts** and **enable interrupts**. Now, the handler should save the return address to the original program while the interrupts are still disabled. Once it does that, it can enable interrupts to ensure that the processor does not miss interrupts that are more important. Before leaving the handler, it restores **k0** with interrupts disabled.

Note 1.6.2.1

Note: it isn't always necessary to entertain a second interrupt while working on the first one. The partnership between handler code and the processor hardware determine how to handle multiple interrupts and there are two options:

1. Ignore the interrupt for a while
2. Attend to the new interrupt immediately.

Sometimes this is determined by having multiple interrupt levels of different priority.

1.7. RTI

In order to manage handling interrupts easier, we cannot just use `Enable Interrupts` and `J $k0$`. What if an interrupt happens between the two instructions?

So we define a new instruction: **Return from Interrupt (RTI)**

Definition 1.7.1

Return from interrupt is *atomic*, meaning this instruction executes fully before any new interrupts can occur. This loads PC from K0 and enables interrupts.

1.8. Datapath Details for Interrupts

The new bus also connects the processor to the memory and more importantly other I/O devices.

- There is a wire labeled INT on the bus. Any device that wishes to interrupt the GPU asserts this line. Any number of devices can simultaneously assert this line to indicate their intent to talk to the processor, using a concept called *wired-or* logic

Theorem 1.8.1

The handshake between the processor and the device for interrupts:

1. The device asserts the INT line whenever it is ready to interrupt the processor
2. The processor upon completion of the current instruction checks the INT line (shown as check INT=y/n in the FSM) for pending interrupts
3. If there is a pending interrupt, then the processor enters the INT macro-state and asserts the INTA line on the bus
4. The device upon receiving INTA places it vector on the data bus
5. The processor receives the vector and looks up the entry in the IVT
6. The processor completes the action in the INT macro-state, saving the current PC in k0 and loading the PC with the value from the IVT

1.9. Stack for saving/restoring

How does the handler know which part of memory is to be used as a stack? The interrupt may not even be for the currently running program.

For this reason, we have two stacks: a **user stack** and a **system stack**. Quite often, the architecture may designate a particular register as the stack pointer. Upon entering INT macrostate, the FSM performs *stack switching*.

Stack switching:

1. Duplicate Stack Pointer

- We duplicate **\$sp** to have one for use by the user program and one for use by the system. The state saving in the interrupt handler will use the version of sp based on what mode they are currently in.

2. Privileged mode

- We now introduce a **mode bit** into the processor. Based on the value of the bit, we are either in the user or kernel mode.
- Before returning to the user program, RETI instruction sets the mode bit back to “user” to enable the user program to go back to using the user stack as it resumes its execution.
- The mode bit also makes it so that the new 3 instructions we added to support interrupts cannot be run by any program. We call these **privileged instructions**. An interrupt handler is part of the OS and runs in the kernel mode, so if a user program tries to use these instructions it will lead to an illegal instruction trap.

2. Processor Speed Metrics

The two performance metrics that we are interested in are:

1. Space Metric (memory footprint)

- The overall running time of the program

2. Time metric (execution time)

- The total space the program takes

The ISA of the processor has a bearing on the memory footprint of the program.

```
Execution time = (sum_{j=1}^n CPI_j) * clock cycle time
```

```
Execution time = n * CPI(avg) * clock cycle time
```

Definition 2.1

Instruction frequency is the metric for understanding how often a particular instruction occurs in programs.

Static frequency refers to the number of times a particular instruction occurs in the compiled code. This has an impact on the memory footprint, so if we find an instruction appears a lot in a program, then we may try to optimize the amount of space it occupies in memory by clever coding techniques.

Dynamic frequency refers to the number of times a particular instruction is executed when the program is actually run. So if we find that the dynamic frequency of an instruction is high we may make enhancements to the datapath and control to ensure that the CPI taken for its executing is minimized.

2.1. Benchmarks

Definition 2.1.1

Benchmarks are a set of programs that are representative of the workload for a processor. For example, for a processor used in a gaming console, a video game may be the benchmark program.

Kernels of real programs are used as benchmarks. Kernels are small, focused pieces of code that represent a core computational task or algorithm commonly found in real world applications.

What might cause static frequency and dynamic frequency differ:

- A loop or repeating segment of a program or a branch around code can cause instructions inside the loop/segment to have a higher or lower dynamic frequency (because they get executed more or fewer times) than their static frequency

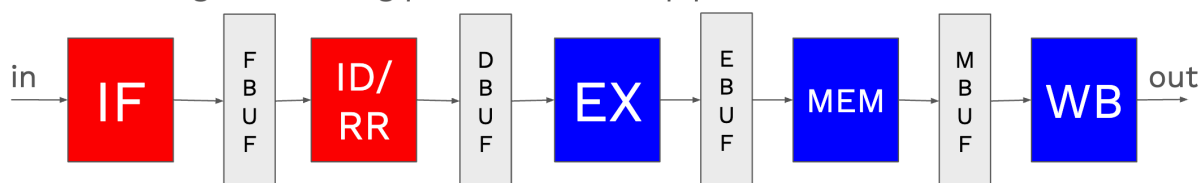
Some ways to evaluate processor performance:

1. **Total execution time**
2. **Arithmetic mean (AM)** is used if you have a set of programs that you want to run at different times but not all at the same time. AM is an average of all the individual program execution times.
3. **Weighted arithmetic mean (WAM)** is useful if you have a know the frequency at which you run the programs.
4. **Geometric mean (GM)** is the p th root of the product of p values. So $P1 = 100$ and $P2 = 1$, $GM = \text{root } 2(100*1) = 10$. This removes the bias from arithmetic mean.
5. **Harmonic mean (HM)** is another useful composite metric which is found by taking the arithmetic mean of the reciprocals of the values, and then taking the reciprocal of that.

3. Pipelining

A radically different approach to improving the processor performance is not to focus on the *latency* for individual instructions but on *throughput*, or the number of instructions executed by the processor per unit time.

Basically latency answers how many clock cycles does the processor take to execute 1 instruction but throughput answers how many instructions can the processor executed in each clock cycle



IF = FETCH

- This stage fetches the instruction pointed to by the PC from I-mem and places it into IR. It also increments the current PC in readiness for fetching the next instruction
- Requires a PC, ALU, and I-MEM

ID/RR = Decode instruction / read register contents

- This stage decodes the instruction and reads the register files to pull out two source operands (more than what may actually be needed depending on the instruction)
- To enable this functionality, the register file has to be dual ported.
- Since this stage contains logic for decoding the instruction as well as reading register file, its called ID/RR
- We need the DPRF

EX = Execute

- This stage does all the arithmetic and/or logic operations that are needed for processing the instructions

MEM = fetch / store memory operands

- This stage either reads from or write to the D-MEM for LW and SW instructions, respectively. Instructions that do not have a memory operand will not need the operations performed in this stage.
- Uses D-MEM

WB = write to register

- This stage writes the appropriate destination register (Rx) if the instruction requires it. Instructions in LC-2200 that require writing to a destination register include arithmetic and logic operations as well as load.
- DPRF

Every stage works on the partial results generated in the previous clock cycle by the preceding stage.

Note 3.1

We need buffer register to store / isolate stage state, noted as **DBUF** for decode buffer. Basically since each stage works on different instructions, once a stage has completed its stage it places the results in a buffer to be picked up in the next clock cycle.

In the steady state, there are 5 instructions in different stages of processing in the pipeline, which makes the CPI effectively 1.

The key points to note in designing a pipeline-conscious architecture are:

- Need for a symmetric instruction format (the locations of register specifiers, sizes, positions of offsets, etc all have to be in the same location)
- Need to ensure equal amount of work in each stage, leading to optimal clock cycle time

Definition 3.1

Data hazards occur when data is changed in one stage of the pipeline, potentially affecting instructions in other stages.


There are 3 kinds of data hazards:

- Read after Write (RAW): reading a value that sequentially just got written
- Write after Read (WAR): writing a value that sequentially just for read
- Write after Write (WAW): write a value that sequentially just got written as well

If its within 4 instructions any of these can happen(?) like I1 -> I4

Flow dependence


$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_5 \leftarrow r_3 \text{ op } r_4$



Read-after-Write
(RAW)

Anti dependence


$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_1 \leftarrow r_4 \text{ op } r_5$



Write-after-Read
(WAR)

Output-dependence

$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_5 \leftarrow r_3 \text{ op } r_4$
 $r_3 \leftarrow r_6 \text{ op } r_7$



Write-after-Write
(WAW)

Definition 3.2

Control Hazards occur as a result of instructions executing non-sequentially.

One example is using a BEQ instruction, changing the value of PC but already having assumed the branch wasn't taken.

The solution: flush instructions out of the pipeline if we are wrong about a branch being taken/not taken.