CS 2200 Exam 1 Cheatsheet

Krish Katariya

Last updated: **September 11, 2024**

Contents

1. Architecture vs Organization

- Depending on cost/performance, several implementation of the same architecture can be possible
    - Ex server processors for higher demand and cheaper consumer processors, leading to a **family** of processors adhering to a particular architecture description

Architecture

Architecture includes the **instruction set architecture** which is the interface between software and hardware

- It defines how data is formatted, addressing modes, and the overall functionality of the machine (how arithmetic operations happen, memory access, branching, etc)

Architecture is more *abstract* and focuses on system design principles

Organization

Organization is focused on the actual hardware implementation details like how the control unit, ALU, registers, memory hierarchy, etc, are structured. It also covers other information such as clock speed etc.

Organization refers to how the ISA is realized in terms of physical component and circuits

- Parallel hardware impacts Architecture vs Organization
- Maintain compatibility for legacy software

2. Addressing Modes

- We cant use direct memory addresses since the number of bits to represent addresses in an instruction are very large.

Definition 2.1

Base+offset mode lets us compute a memory address as the sum of the contents of a register in the processor (base register) and an offset (contained directly in the offset)

Base+offset addressing mode can be used to support structured data types in high level languages.

Definition 2.2

Base+index addressing mode lets us calculate the effective memory address as the sum of two registers.

This is especially useful when looping through an array or different structures.

Definition 2.3

PC-relative addressing is used to compute effective addresses especially with conditional branching such as BEQ.

to complete

3. Stack Frames & Calling Convention

Theorem 3.1

Use: **Program Data**
- Registers **s0-s2** are the caller's "save" registers
- Registers **t0-t2** are the temporary registers
- Registers **a0-a2** are the parameter-passing registers
- Register **v0** is used for the return value

Use: **Bookkeeping**
- Register **ra** is used for the return address
- Register **at** is used for jump target address
- Register **sp** is used as a stack pointer

```
lea $sp, stack #get the address of stack value
lw $sp, 0($sp) #put it into sp

addi $sp, $sp, -1 # storing t0
sw $t0, 0($sp)

addi $sp, $sp, -1 # storing t1
sw $t1, 0($sp)

addi $sp, $sp, -1 # storing t1
sw $t2, 0($sp)

addi $sp, $sp, -1 #space for additional parameters values
sw RX, 0($sp) # store something in additional parameter values
addi $sp, $sp, -1 # space for additional return value
sw RX, 0($sp)
```

```
addi $sp, $sp$, -1 # space for previous return address
sw $ra, 0($sp$)

lea $at, target

JALR $at, $ra # JALR TO THING

# BACK TO CALLER NOW
lw $ra, 0($sp) # restoring previous return address to ra
addi $sp, $sp, 1

lw $t0, 0($sp) # fetches our return value
addi $sp, $sp, 1

addi, $sp, $sp, 1 # removing additional parameters

addi, $sp, $sp, 1 # restoring t0
lw $t0, 0($sp$)

addi, $sp, $sp, 1 # restoring t1
lw $t1, 0($sp$)

addi, $sp, $sp, 1 # restoring t2
lw $t2, 0($sp$)


target:

addi $sp, sp, -1 # space for frame pointer
sw $fp, 0($sp)

addi $fp, $sp, 0 # sets current fp to the current sp

addi $sp, $sp, -3 #making space to save s0-s2
sw $s0, 0($sp)
sw $s1, 1($sp)
sw $s2, 2($sp)

addi $sp, $sp, -1 # allocating space for local variable.
# The program runs here

TEARDOWN
addi $sp, $sp, 1 # popping local variables

lw $s0, 0($sp$) # loading s0
addi $sp, $sp, 1

lw $s1, 1($sp$) # loading s1
addi $sp, $sp, 1

lw $s2, 2($sp$) # loading s2
addi $sp, $sp, 1

lw $fp, 0($sp) # loading old frame pointer
addi $sp, $sp, 1
```

```
jalr $ra, $zero # JALR jumps to the target and puts PC+1 into $ra. This throws away the return
address
```

The stack grows towards lower memory addresses. This means we decrement then push, or pop then increment.

Steps for **calling convention build-up**:

1. Caller saves any of registers t0-t2 on the stack (if it needs the value in them upon return)
2. Caller places the parameters in a0-a2 (using the stack for additional parameters if needed)
3. Caller allocates space for any additional return values on the stack
4. Caller saves previous return address currently in **ra** (in case we recurse again)
5. Caller executes JALR **at**, **ra**
6. Callee saves any of registers s0-s2 that it plans to use during its execution on the stack
7. Callee allocates space for any local variables on the stack

Steps for **teardown**:

1. Prior to return, Callee restores any saved s0-s2 registers from the stack
2. Callee executes jump to ra, no change to stack
3. Upon return, caller restores the previous return address to **ra**
4. Caller stores additional return values as desired
5. Upon return, caller moves stack pointer to discard additional parameters
6. Upon return, caller restores any saved t0-t2 registers from the stack

Frame Pointer

We tend to use a **frame pointer** which keeps track of an address of a known point in the *activation record* for the function and doesn't change during execution of the function (while stack pointer changes based on number of extra arguments and local variables etc).

For example in the following code

```
int a, b;

if (a > b) {
  int c = 1;
  a = a + b + c;
}
```

When you do `int c = 1`, the stack pointer goes up one because we added a new local variable on the stack. Keeping track of all of these variables would be very hard which is why frame pointers make things easier.

4. LC 2200/4000 ISA

LC-2200

The LC-2200 is a 32-bit register-oriented little-endian architecture with a fixed length instruction format. There are 16 general purpose registers as well and all addresses are word addresses.

The LC-2200 supports four instruction formats.

4.1. Instructions:

R-type instructions(add, nand)

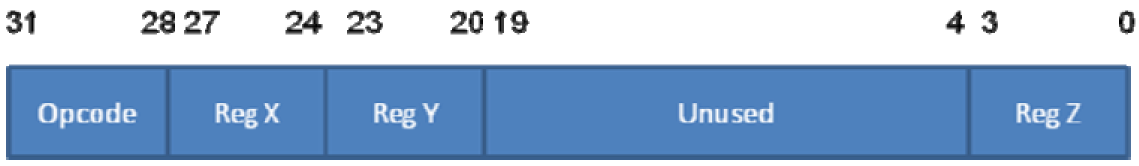
bits 31-28: opcode

bits 27-24: reg X

bits 23-20: reg Y

bits 19-4: unused

bits 3-0: reg Z



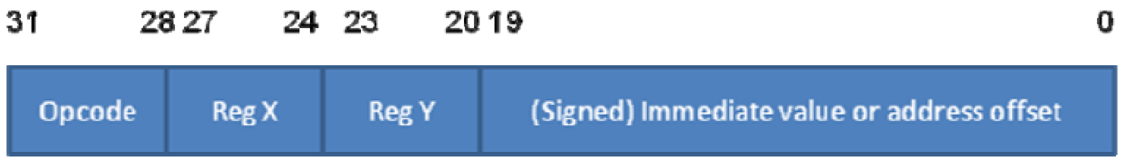
I-type instructions(addi, lw, sw, beq):

bits 31-28: opcode

bits 27-23: Reg X

bits 23-20: Reg Y

bits 19-0: Immediate value / address offset



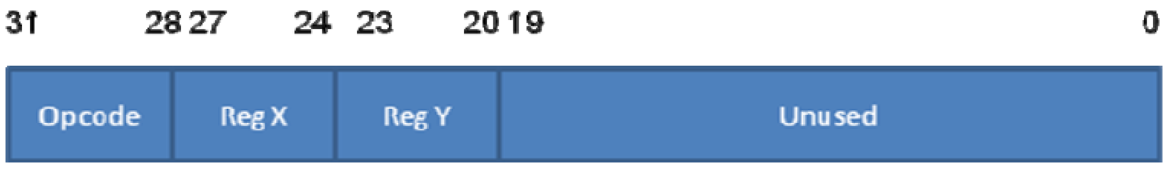
J-type instructions(jalr):

bits 31-28: opcode

bits 27-24: reg X (target of the jump)

bits 23-20: reg Y (link register)

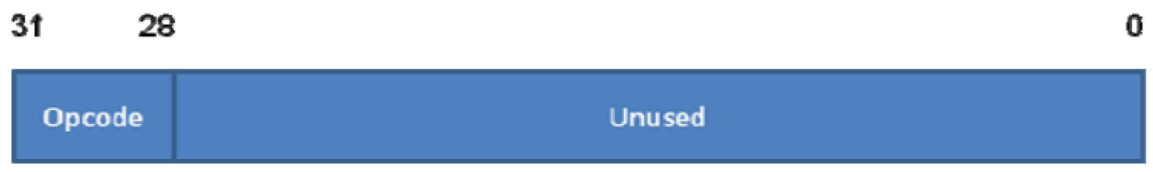
bits 19-0: unused (all zeroes)



O-type instructions(halt):

bits 31-28: opcode

bits 27-0: unused



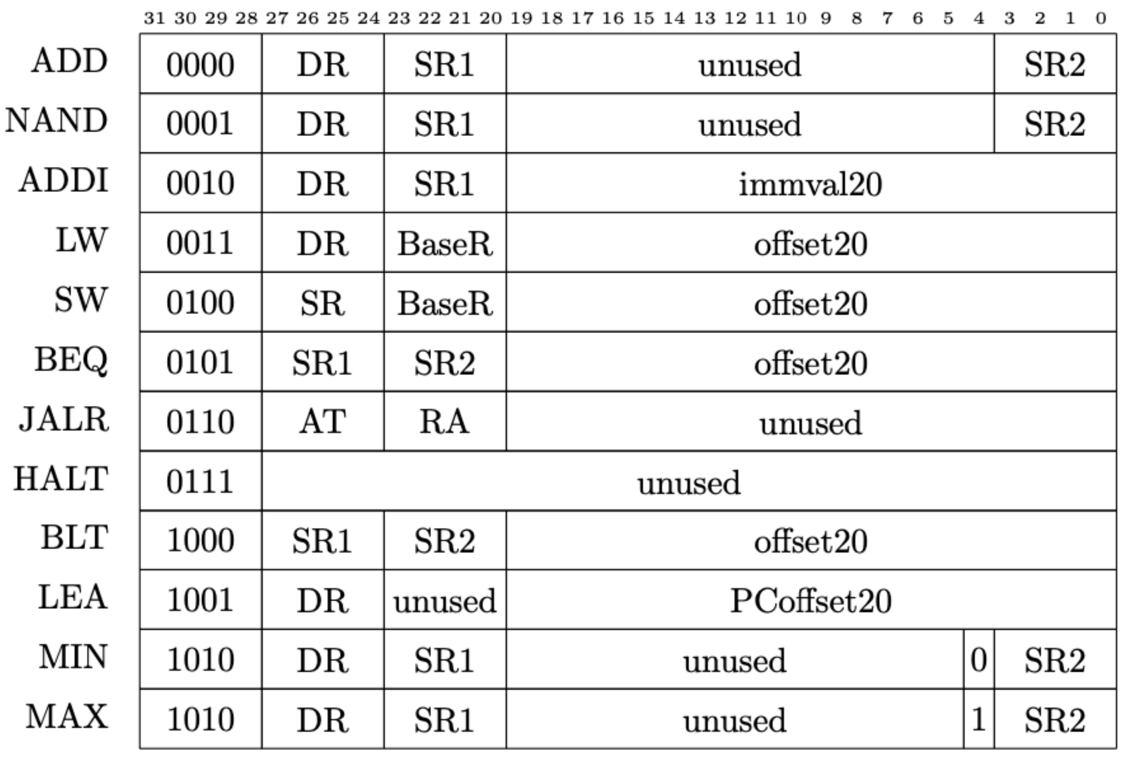
4.2. LC-2200 Register Set

Reg Number	Name	Use	Callee Save?
0	$zero	Always zero (by hardware)	N/A
1	$at	Target Address	N/A
2	$v0	Return value	No
3	$a0	Argument	No
4	$a1	Argument	No
5	$a2	Argument	No
6	$t0	Temporary	No
7	$t1	Temporary	No
8	$t2	Temporary	No
9	$s0	Saved register	Yes
10	$s1	Saved register	Yes
11	$s2	Saved register	Yes
12	$k0	Reserved for OS/traps	N/A
13	$sp	Stack pointer	No
14	$fp	Frame pointer	Yes
15	$ra	Return address	No

- Register **at** keeps track of the target address of a jump. Pseudo-instructions generated by the assembler may also be used.
- **v0** is where you keep track of any returned value from a subroutine call.
- **a0-a2** stores the function/subroutine arguments. These registers can be trashed.
- **t0-t2** stores temporary variables that the caller must save if they want to use these values.
- **s0-s2** are saved registers that the caller may assume the subroutine never tampered with. If these are tampered with then the callee should save them on the stack and restore them.
- **k0** is used for handling interrupts.
- **sp** is used to keep track or the top of the stack.
- **fp** is used to be the anchor point of an activation frame.
- **ra** is used to store the address a subroutine should return to when it is finished executing.

4.3. Instructions

Table 2: LC-4000 Instruction Set



	31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8 7 6 5	4	3 2 1 0
ADD	0000	DR	SR1	unused		SR2
NAND	0001	DR	SR1	unused		SR2
ADDI	0010	DR	SR1	immval20		
LW	0011	DR	BaseR	offset20		
SW	0100	SR	BaseR	offset20		
BEQ	0101	SR1	SR2	offset20		
JALR	0110	AT	RA	unused		
HALT	0111	unused				
BLT	1000	SR1	SR2	offset20		
LEA	1001	DR	unused	PCoffset20		
MIN	1010	DR	SR1	unused	0	SR2
MAX	1010	DR	SR1	unused	1	SR2

4.3.1. Conditional Branching

- BRQ and BLT offer the ability to branch upon a certain condition being met using comparison operators that compare the value of two source registers.
- For MIN, if SR1 < SR2, we branch to a series of microstates that stores the value in SR1 to DR. Otherwise, we branch to a series of microstates that store the value in SR2 to DR. Max is the opposite of this.

5. Big vs Little Endian

Definition 5.1

Big-endian places the *most significant bit* first (at the lowest memory address). In this format, the byte representing the largest part of the number comes first

For example with: **0x12345678**

Address	**Byte**
0x1000	12
0x1001	34
0x1002	56
0x1003	78

Definition 5.2

Little-endian places the *least significant bit* first (at the lowest memory address). In this format, the byte representing the smallest part of the number comes first.

For example with: **0x12345678**

Address	**Byte**
0x1000	78
0x1001	56
0x1002	34
0x1003	12

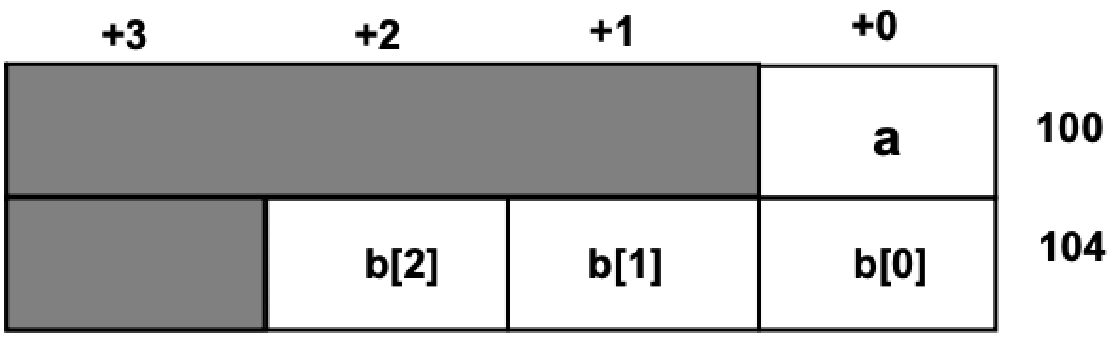
6. Packing

- A compiler might try to **pack** operands of a program in memory to conserve space. This is mostly useful when data structures consist of variables with different **granularities** (such as int, chat, etc) and if the architecture supports multiple levels of precision in operands.

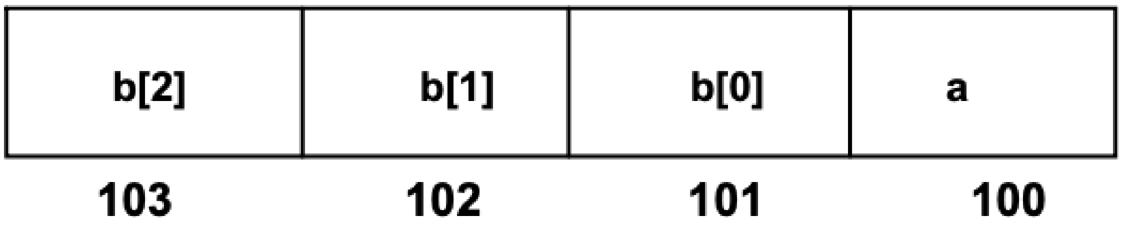
For example there are two ways to organize this struct:

```
struct {
  char a;
  char b[3];
}
```

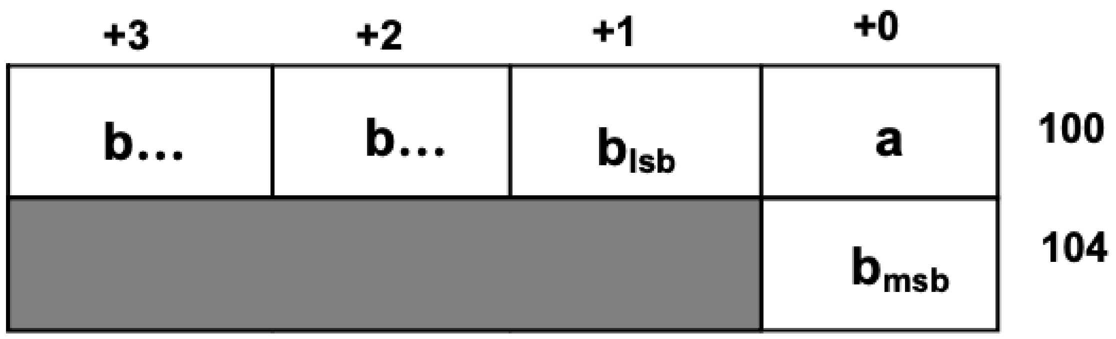
Unpacked:



Packed:



While packing can save space it is not always the best option for all processors. For example this packing structure would cause us to load two words from memory which is inefficient.



For this reason architectures usually require word operands to start at word addresses, which is known as **alignment restriction** of word operands to word addresses. This means that `ld r2, address` would be an illegal instruction of the address is not on a word boundary (1000, 1004, etc)

While packing can save time it might lead to less efficiency in the point of view of time to access operands.

7. LC 2200/4000 Datapath & Control Signals

7.1. Edge Vs Level Triggered

Definition 7.1.1

Edge Triggered Logic means that output changes can only happen at the exact second that that clock switches from 0 to 1 or from 1 to 0. For this course I think we assume *positive edge triggered logic* in most cases.

On the other hand, **level triggered logic** means a chance can happen as long as the clock signal is 1.

Definition 7.1.2

Clock Width is defined by taking the potential paths of signal propagation in every clock cycle. Then we choose the clock width to be greater than the worst case delay in the entire datapath.

Every combination logic element (like the ALU) has latency for propagating a value from its input to the output known as propagation delay. Similarly there is a latency for accessing values from a register.

Types of delays:

- Register input
  - Setup (before clock edge): this is the minimum time that the data input into a register must be stable *before* the clock edge arrives. The system needs to ensure that the data is "ready" to be latched into the register when the clock triggers the storage operation.
  - Hold (after clock edge): after the clock edge, the data must remain stable for a short period of time to ensure the register correctly captures the value.
- Output stable
  - After a register captures the data, its output doesn't immediately reflect the newly captured data. The system needs a short amt of time to stabilize the output before it can eb used. This is important so ALU/ memory get correct output.
- Propagation
  - Wire: This refers to the time it takes for signals to travel from one component to another along a wire.
- ALU operation: the time it takes for an ALU to perform an iteration

Theorem 7.1.1

Level-triggered Devices

- Memory read
- Register file read
- ALU
- Muxes
- Decoder

Edge-triggered devices

- Memory write
- Register write
- Other register wires

- If we tried to make ad-hoc connections among all the main datapath elements to get each instruction executed we would have to create a path from the memory to the register file we would have way too many wires, which is why we use **single bus design**. Other datapath designs may use **dual bus design** which usually requires a dual register file.

7.2. ROM Design

- The control unit of a processor is a sequential logic circuit as well, which we represent as an FSM with Fetch -> Decode -> Execute -> Fetch. There are several possible alternate designs to generate the control signals, all of which are the hardware realization of the FSM abstraction for the control unit of the processor.

Definition 7.2.1

- A simple design is knowing what **state** the processor is in. Since each **macro-state** has its own **micro-states**, we can keep track of the states for each microstate and create a register that holds the current state based on what microstate is running. This means at any instance the register shows the current state of the processor.
  - If we use the state as the index into a table, we can have that table then keep track of what control signals need to be propagated on.

- There are two possibles issues that arise with this method.
  - The first issue is **time**, because to generate the control signals in a particular clock cycle an address has to presented to the ROM and wait for the **access time** of the ROM.
  - The second issue is **space** because the horizontal microcode issue is not space efficient compared to vertical microcode (?) methods
- The M bit is on when we want to read the OPcode from the IR to modify the NextState address
  - If we want to set or unset the M bit we have to do this in the **previous clock cycle** since we save M in the register
- The M bit is on in fetch3 and throughout the execute macrostate

8. Jump Table

Theorem 8.1

Benefits of conditional branching:
- No Memory access
- Small number of branches = efficiency
- Varied comparison logic / non contiguous values

Benefits for jump table:
- Large number of branches
- Contiguous comparison values
- Efficient clock cycles
- Reusable addresses

9. Interruptions, Exceptions, and Traps

HIGHKEY THIS PART IS A MESS USE SOMETHING ELSE

- **Synchronous events**: occur at well defined points aligned with activity of the system
  - Making a phone call
- **Asynchronous events**: occur unexpectendly with respect to ongoing activity of the system
  - Receiving a phone call

Type	Sync/Async	Source	Intentional?	Examples
Exception	Sync	Internal	Yes and No	Overflow, Divide by zero, Illegal memory address, Java exception mechanism
Trap	Sync	Internal	Yes and No	System call, Software interrupts, Page fault, Emulated instructions
Interrupt	Async	External	Yes	I/O device completion

9.1. Interrupts

Definition 9.1.1

An **interrupt** is the mechanism by which devices catch the attention of the processor. This is an unplanned discontinuity for the currently executing program.

These are asynchronous events usually produced by IO devices which must be handled by the processor by interrupting executing of currently running processes.

9.2. Exceptions

Definition 9.2.1

An **exception** may unintentionally perform certain illegal operations (such as divide by zero) or follow an executing path unintended in the program specification. In such cases, once again it becomes necessary to discontinue the original sequence of instruction executing of the program and deal with this kind of unplanned discontinuity.

Synchronous events usually associated with software requesting something that hardware cant perform (illegal addressing)

9.3. Traps

Definition 9.3.1

People often make *system calls* to read/write files or for other such services from the system, and these calls are like procedure calls but need special handling because the user program does not have the authority to do this on its own.

Traps allow the program to fall into the operating system, which will then decide what the user program wants. These are Synchronous events produced by the special instructions to allow secure entry into OS code.

There are four things that make discontinuities tricky

1. They can happen anywhere during the instruction executing.
2. The discontinuity is unplanned for and quite possibly completely unrelated to the current program in execution
3. At the point of detecting the program discontinuity, the hardware has to determine the address of the handler to transfer control from the currently executing program to the handler
4. Since the hardware saved the PC implicitly, the handler has to discover how to resume normal program execution

New internal processor register **ETR**: will contain unique number stashed by the hardware to indicate the type of discontinuity.

Interrupt vector table is a table of all the possible traps and the pointer to the code that handles them.

- In the case of traps and exceptions, the hardware generates this vector internally. We introduce a exception/trap register **ETR**. Upon an exception or a grap, the unique number associated with that exception or trap will be placed in ETR.
- Used by the processor to index into the IVT to get the handler address.

INT is a new flag when it equals 0 in execute go to fetch else handle the interruption before going to fetch.

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 during program execution (such as arithmetic overflow and divide by zero).

2. The operating system may define its own set of exceptions (software interrupts)

and traps (system calls) and specify the numbers (vector values) associated with them.
3. The operating system sets up the IVT at boot time with the addresses of the

handlers for dealing with different kinds of exceptions, traps, and interrupts.
4. During the normal course of execution, the hardware detects exceptions/traps and

stashes the corresponding vector values in ETR.
5. During normal course of execution, the hardware detects external interrupts and

receives the vector value corresponding to the interrupting device.
6. The hardware uses the vector value as an index into the IVT to retrieve the code.

The modified FSM includes a new macrostate for handling interrupts. The FSM 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. If there is no interrupt pending (INT = n) then the next instruction executing resumes by returning to the Fetch macro state.