Georgia Tech Computer Science

CS2200 Systems and Networks Spring 2024

Lecture 7: Control Path (cont'ed)

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

Lecture slides adapted from Bill Leahy, Charles Lively of Georgia Tech

Today's agenda

- Finish up control path
  - Implementation of the Fetch and Decode macro-states
  - Couple more examples of instruction execution steps
  - Micro-sequencer

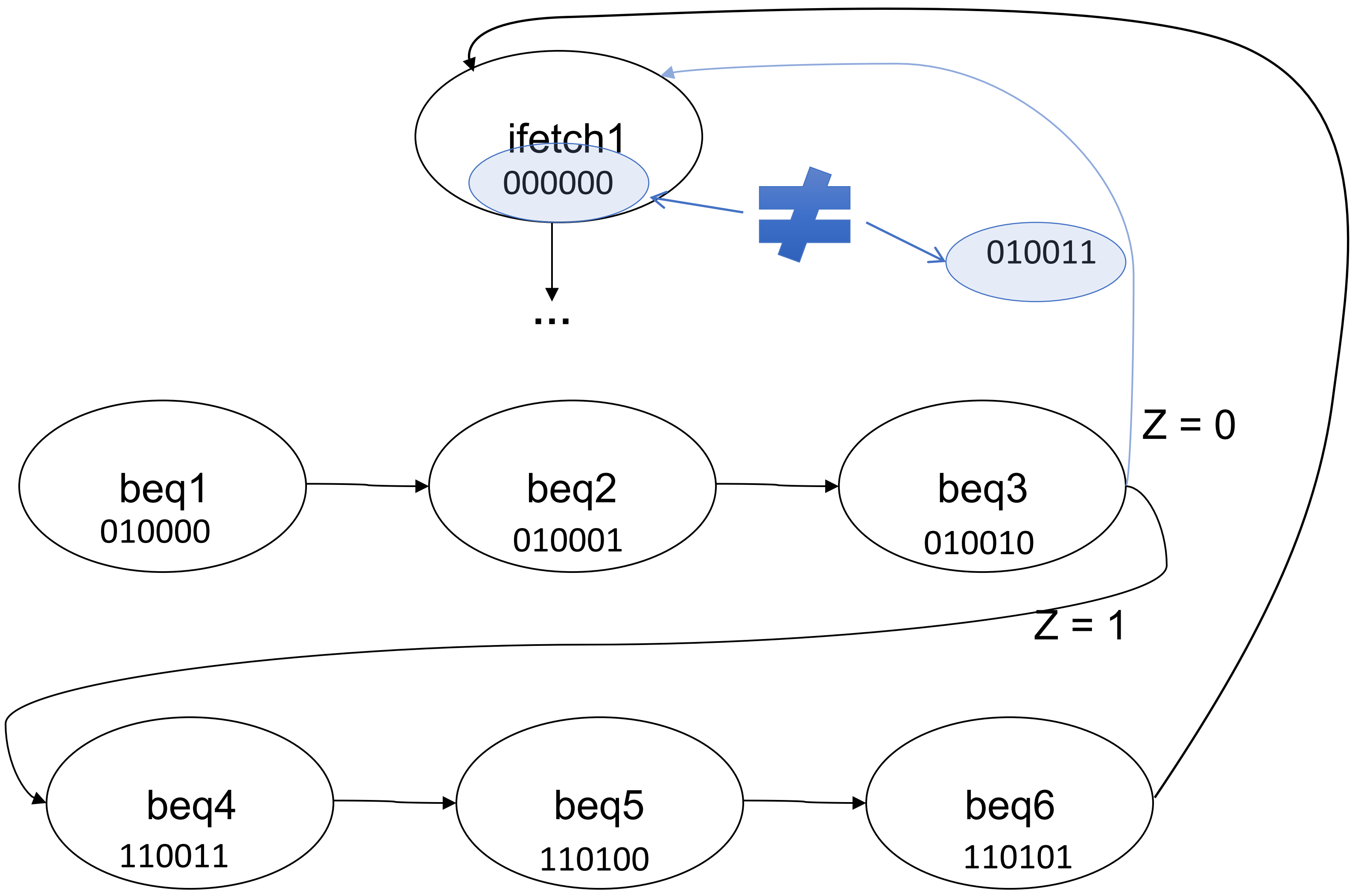
- Start Interrupts, Traps, Exceptions
  - Chapter 4

So We Need to Set T in the Microcode

	Drive Signals					Load Signals						Write Signals					
Current State	PC	ALU	Reg	MEM	OFF	PC	A	B	MAR	IR	Z	ME M	REG	func	Reg Sel	T	Next State
010000			1				1								00		10001
010001			1					1							01		10010
010010		1									1				10	1	10011
110011	1						1									1	10100
110100					1			1								1	10101
110101		1				1								00			00000
010011																	

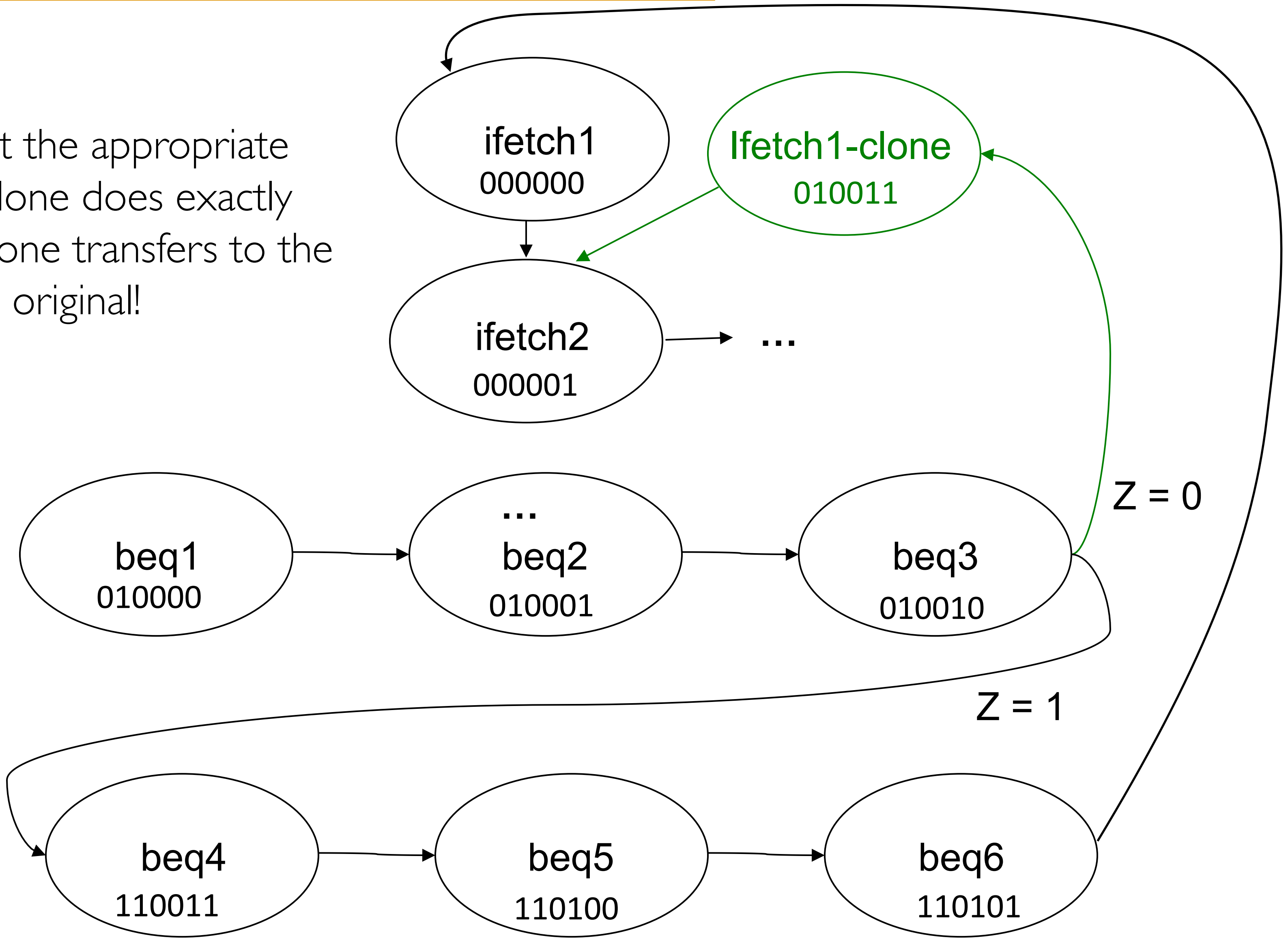
Here we need to fill in the ***contents*** of ROM location ifetch1. Why?

What's the Problem with the Z Branch?

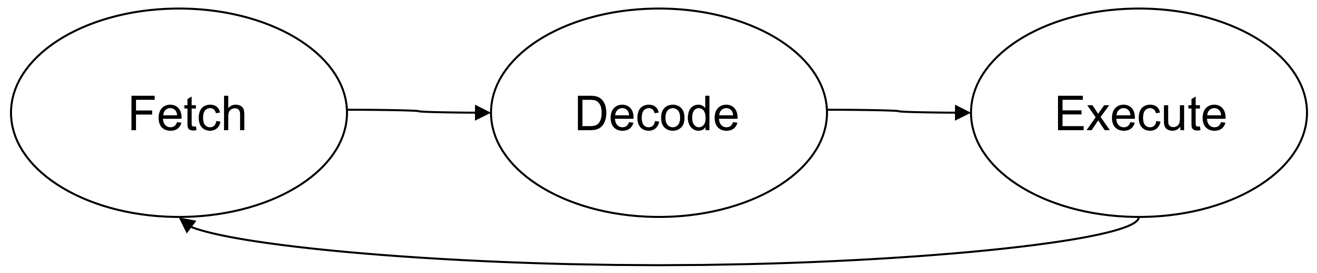


An Old Microcode Trick!

We duplicate ifetch1 at the appropriate ROM address (so its clone does exactly the same thing). The clone transfers to the same next-state as the original!



Back to basic State Diagram


$$\begin{tikzcd}
\text{Fetch} \arrow[r] & \text{Decode} \arrow[r] & \text{Execute} \arrow[ll, bend left]
\end{tikzcd}$$

FETCH macro-state

- Need to do
  - We need to send PC to the memory
  - Read the memory contents
  - Bring the memory contents read into the IR
  - Increment the PC
  - (And decode the opcode by branching to the right execution state)

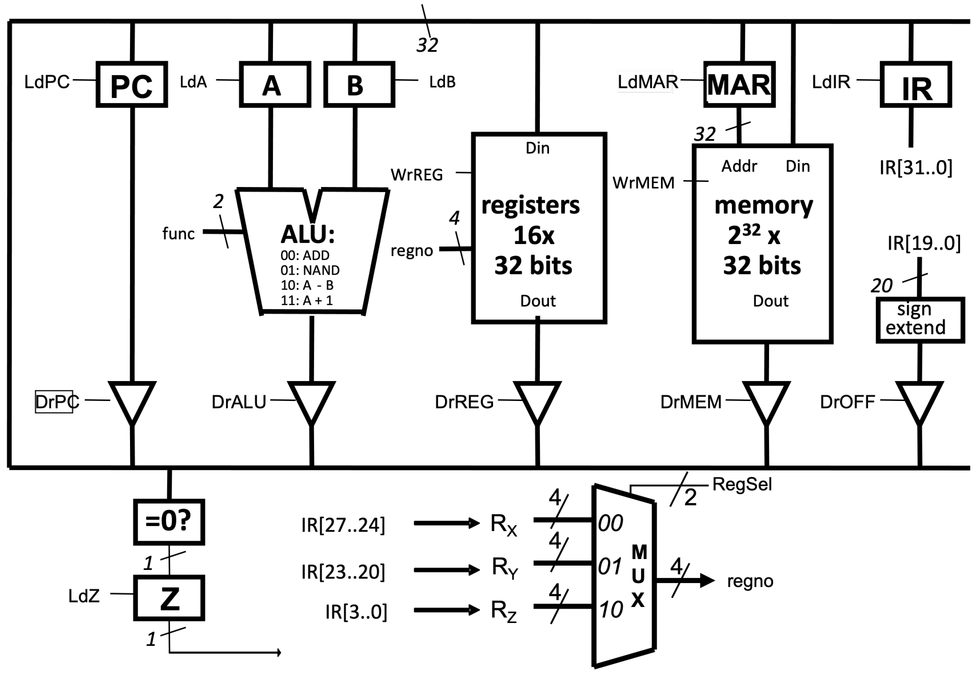
- Microstates to accomplish
  - ifetch1
    - PC → MAR
  - ifetch2
    - MEM[MAR] → IR
  - ifetch3
    - PC → A
  - ifetch4
    - A+1 → PC

Simplify

- ifetch1
  - PC → MAR
  - PC → A
- ifetch2
  - MEM[MAR] → IR
- ifetch3
  - A+1 → PC

FETCH state: Adding in control signals

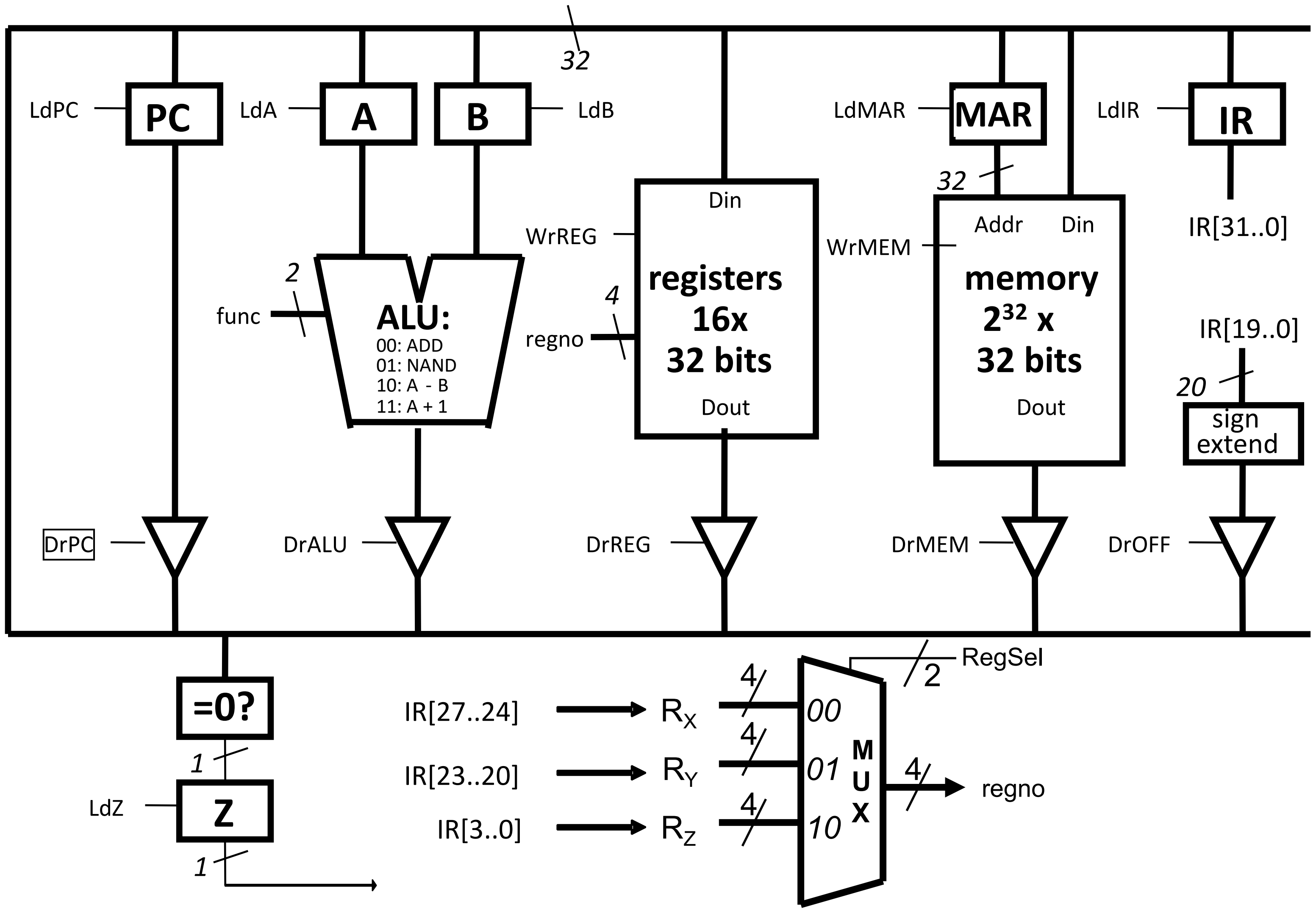
- ifetch1
  - PC → MAR
  - PC → A
  - Control signals needed:
    - DrPC
    - LdMAR
    - LdA



- ifetch2
  - MEM[MAR] → IR
  - Control signals needed:
    - DrMEM
    - LdIR

- ifetch3
  - A+1 → PC
  - Control signals needed:
    - func = 11
    - DrALU
    - LdPC

Before ifetch1

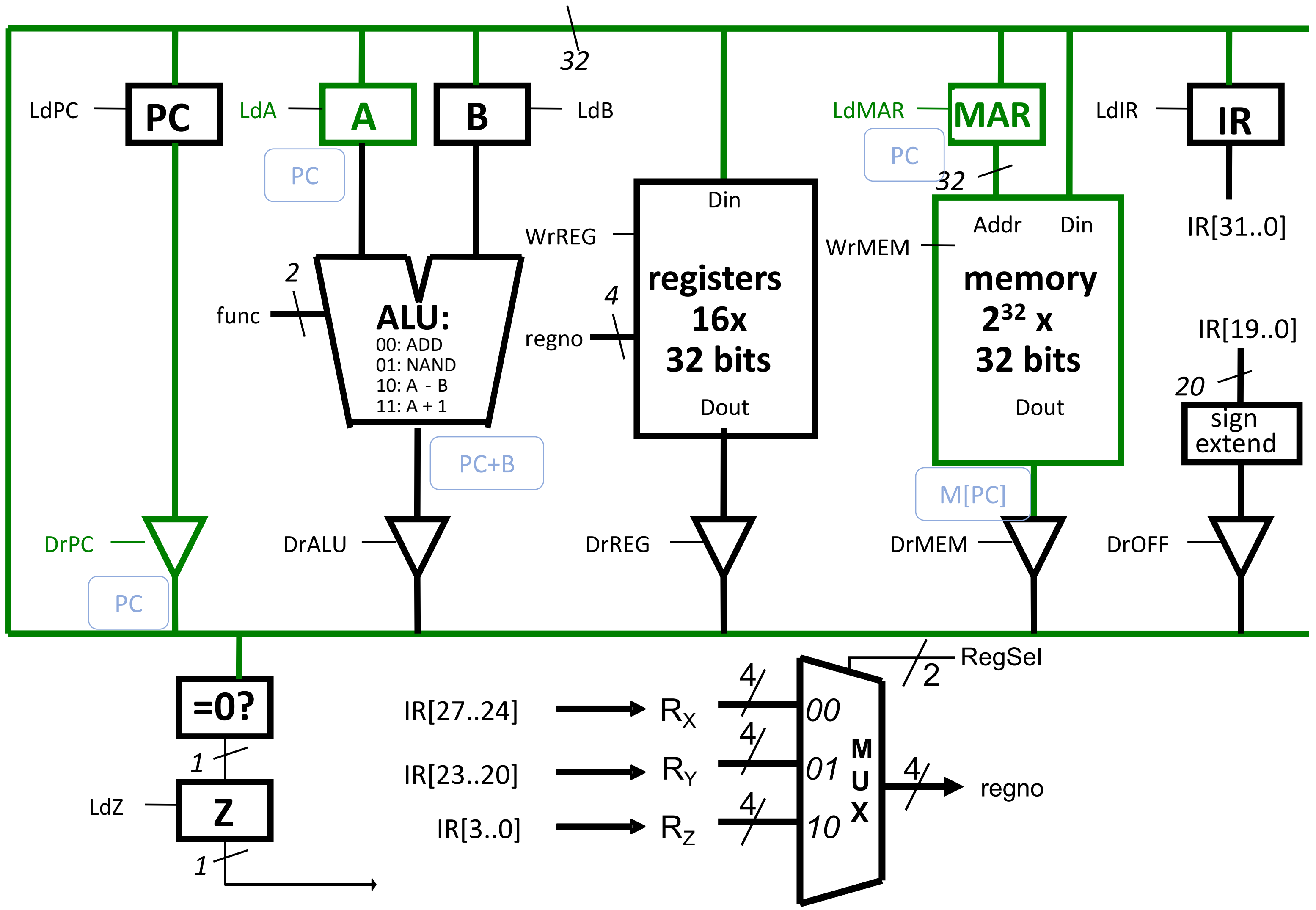


Implementing ifetch1 (end of clock 1)

- PC → MAR
- PC → A

- Control signals needed:

DrPC
LdMAR
LdA
Others=0

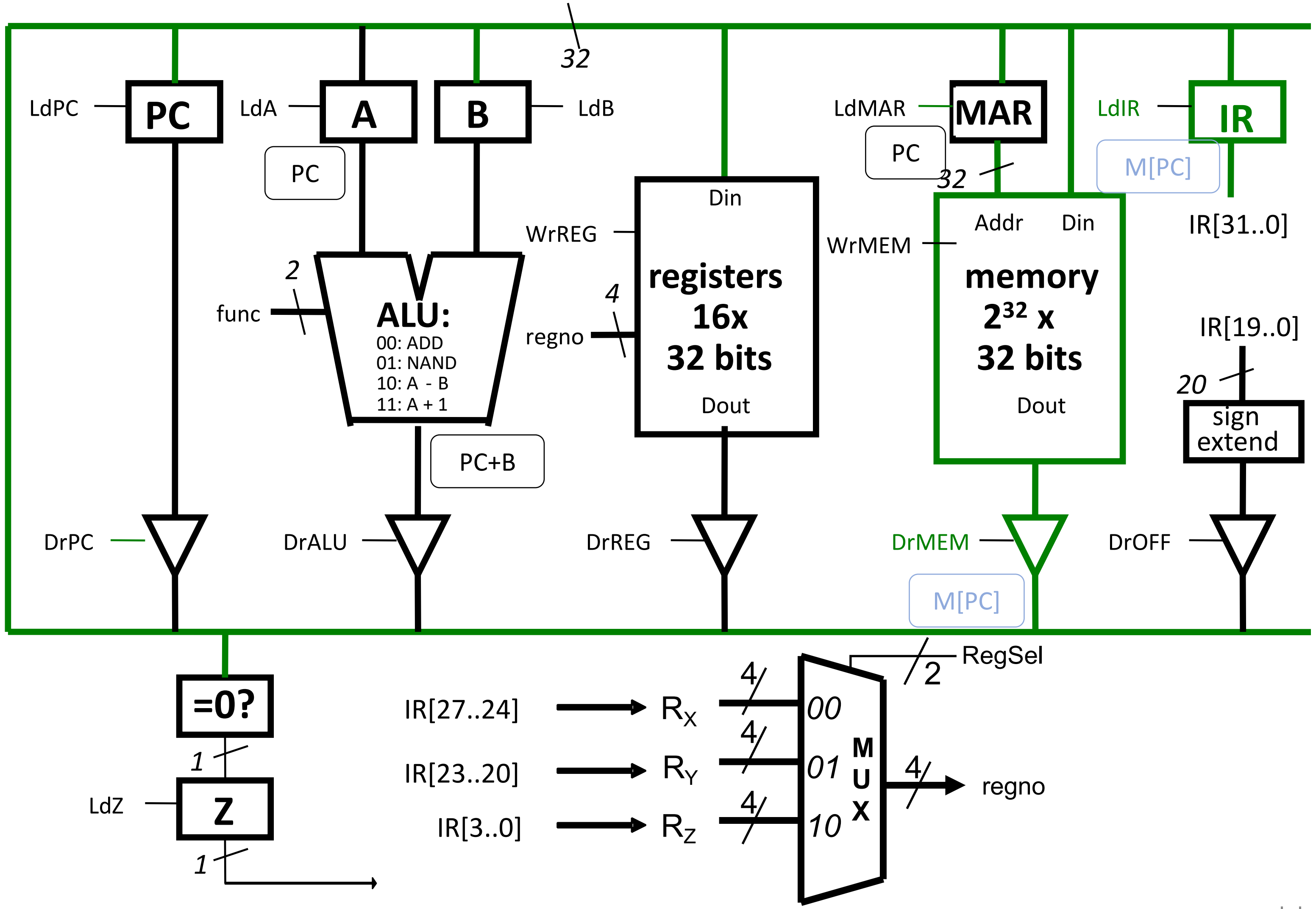


Implementing ifetch2 (end of clock 2)

- MEM[MAR] → IR
- Control signals needed:

DrMem
LdIR

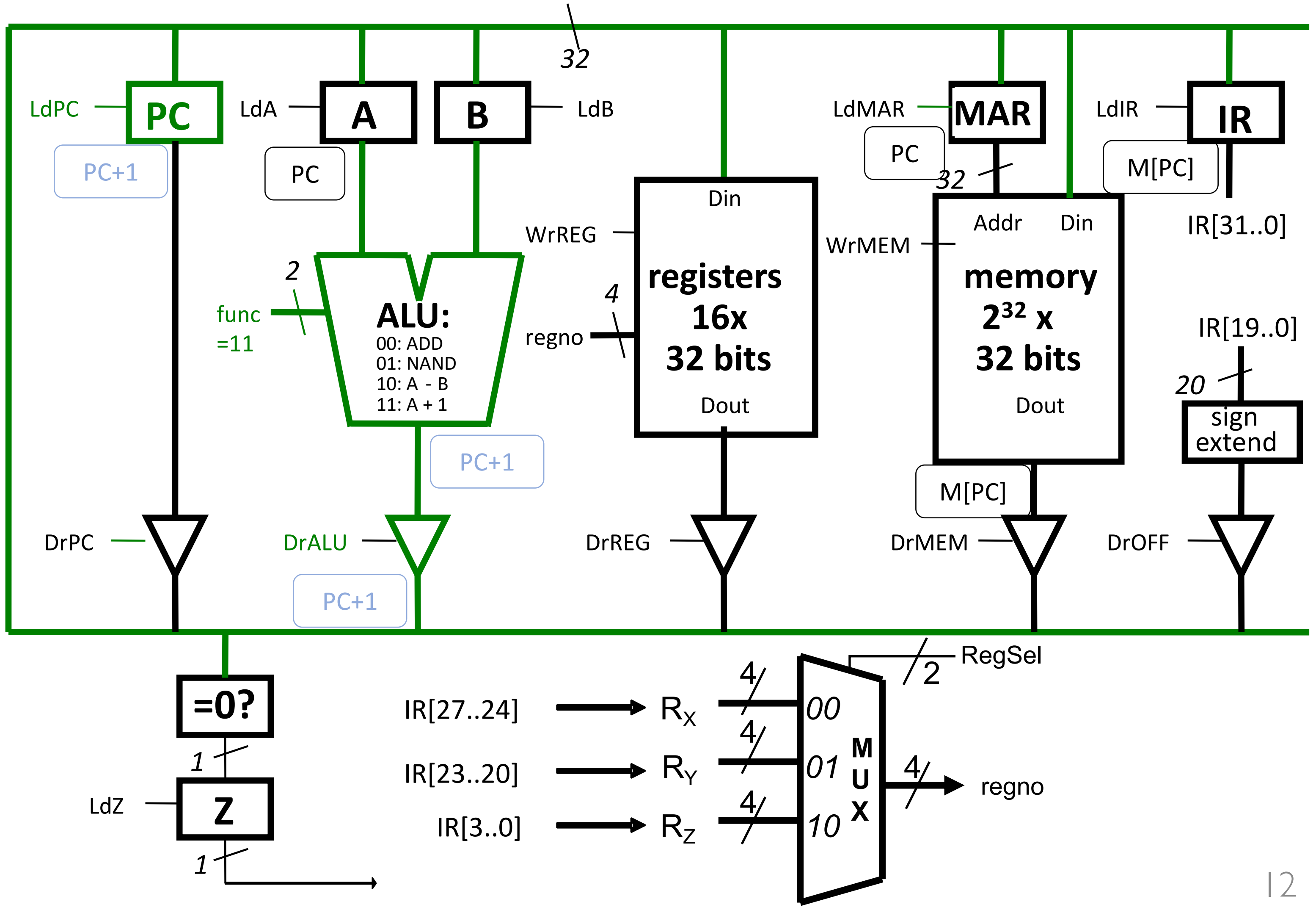
Others=0



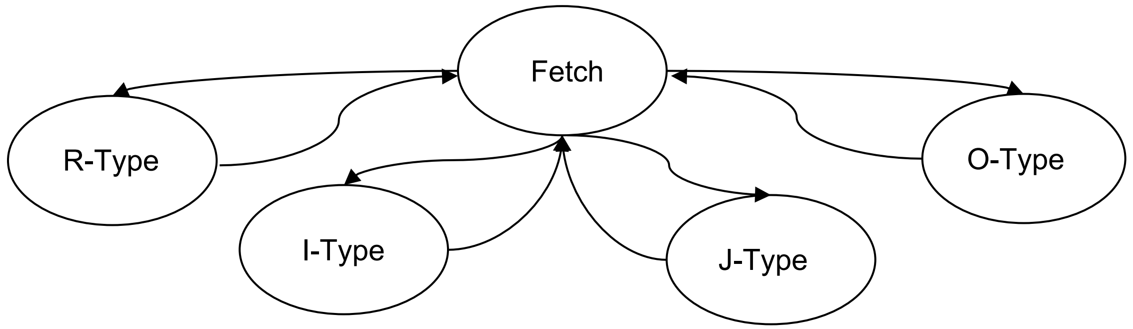
Implementing ifetch3 (end of clock 3)

- A+1 → PC
- Control signals needed:

func=11
DrALU
LdPC
Others=0

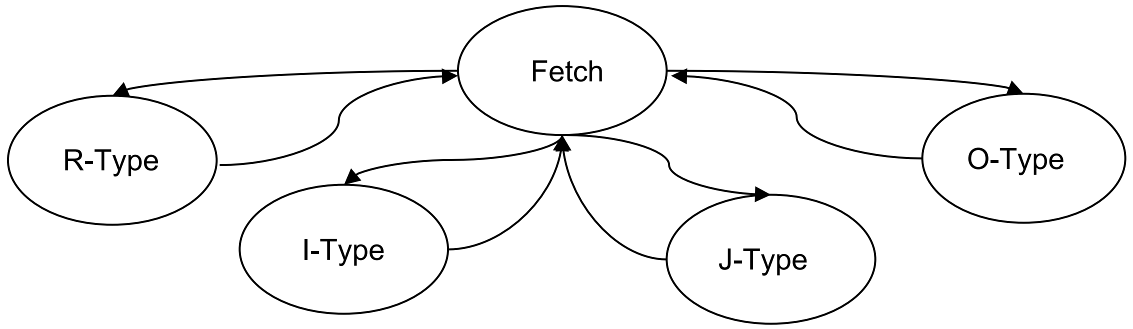


DECODE State



- Decode is a MULTIWAY branch!
- We can't encode this in Next State!
- Actually, we can… let's reuse the same trick we applied for BEQ!

DECODE State



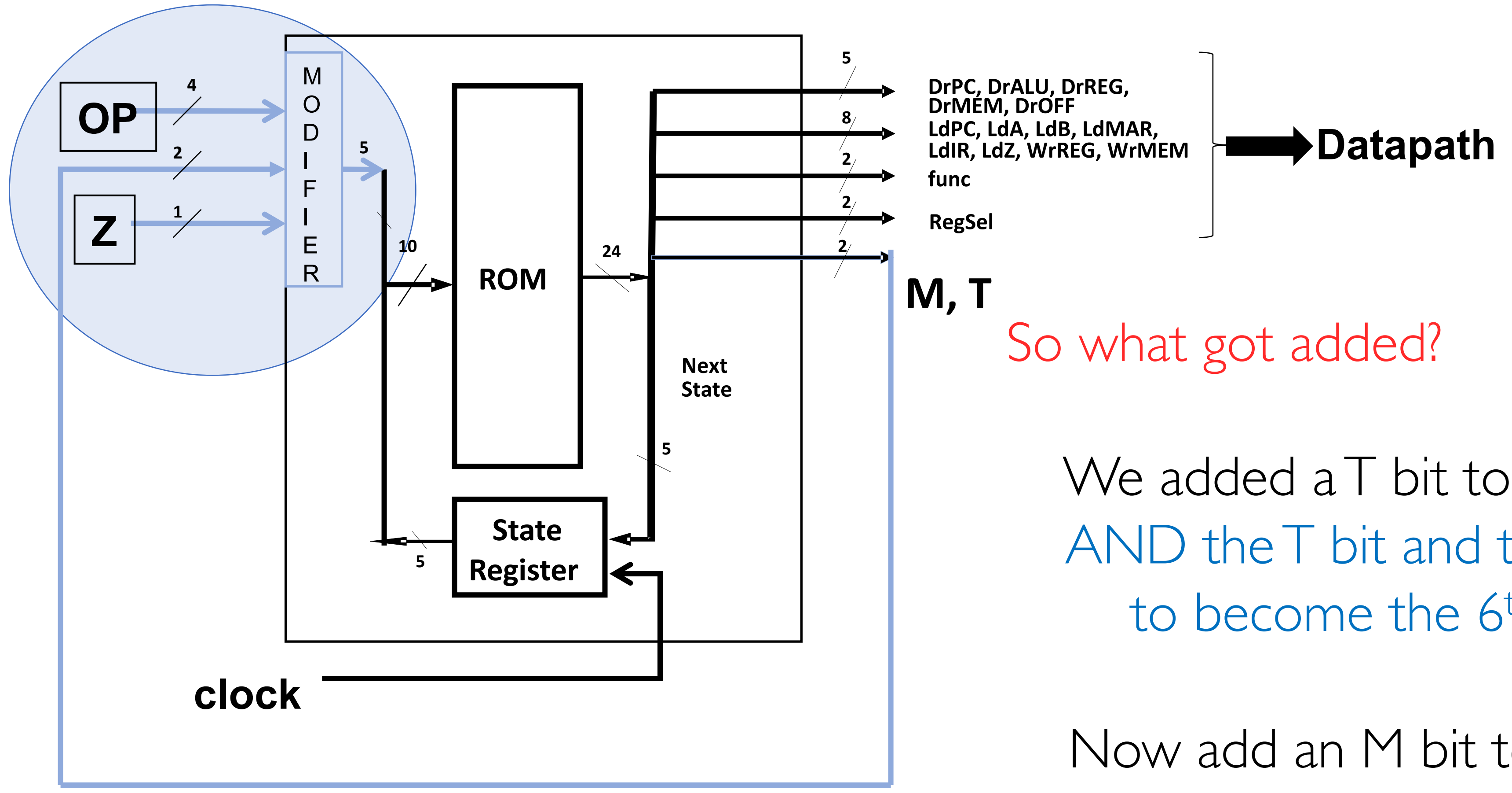
- On the last step of ifetch, we'll set the top 4 bits of our ROM address to the opcode that's in IR[31:28]!
- OK. How do we do that?

Let's extend the Control Unit again

- We expanded the ROM address from 5 to 6 bits for BEQ
- Let's extend it from 6 to 10 bits and use the opcode as the top 4 bits of the ROM address

- I.e., if the opcode is 0010 and next-state is 000011, then if we use the opcode bits, we would use 0010 000011 as the next state so the microcode to execute 0010 would start at that address
- This gives us a many-way branch!

Updated Control Unit

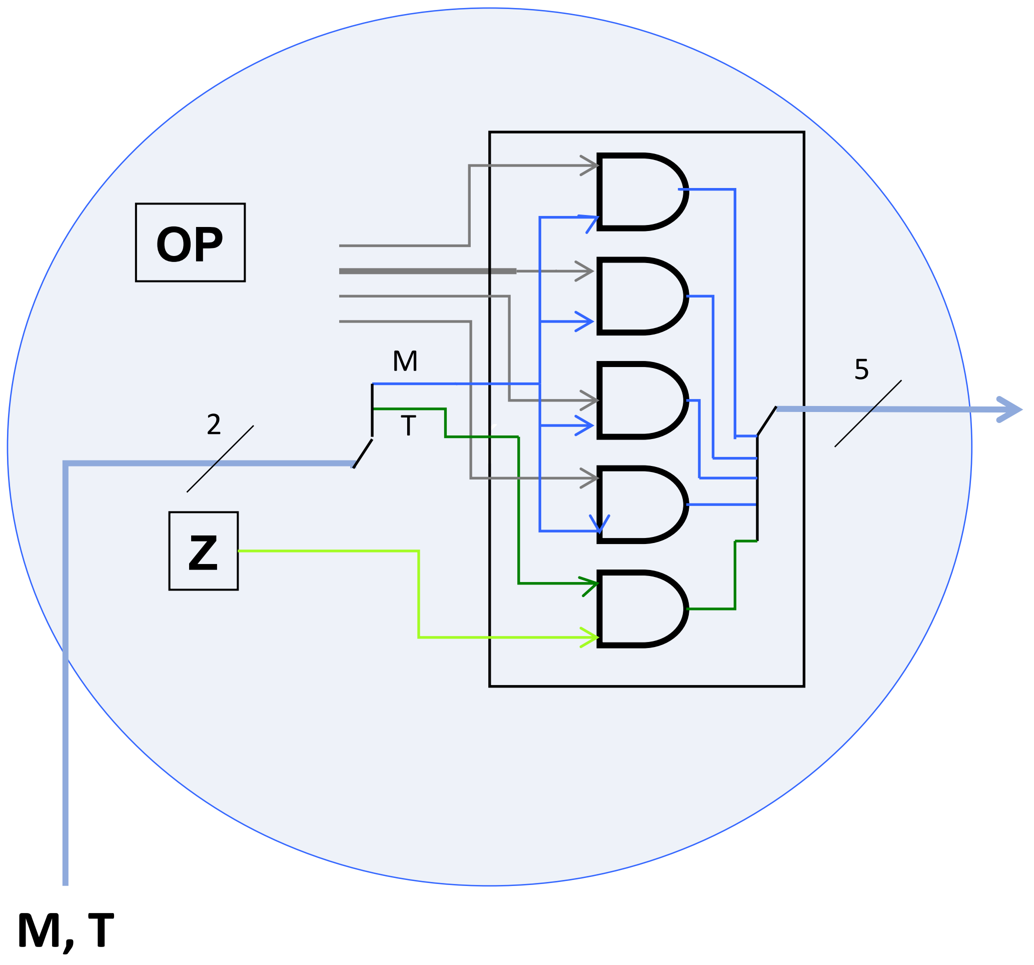


So what got added?

We added a T bit to the ROM for BEQ
AND the T bit and the Z bit
to become the 6th address bit

Now add an M bit to the ROM
AND the M bit and the OP from bits IR[31:28]
to become the 10th-7th address bits

What's in MODIFIER?



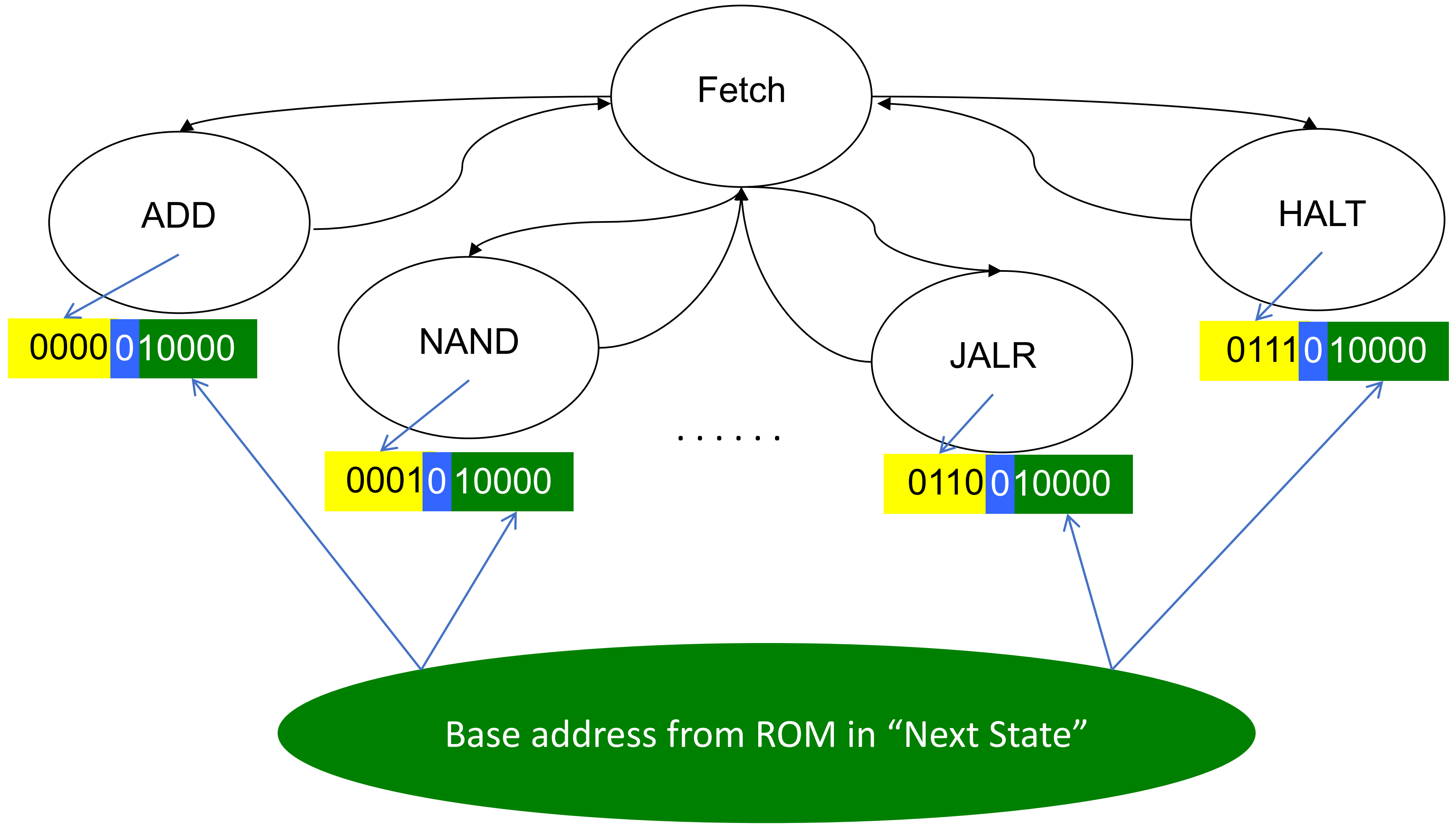
Top 5 bits of Next State

Let's Encode the 3 ifetch States

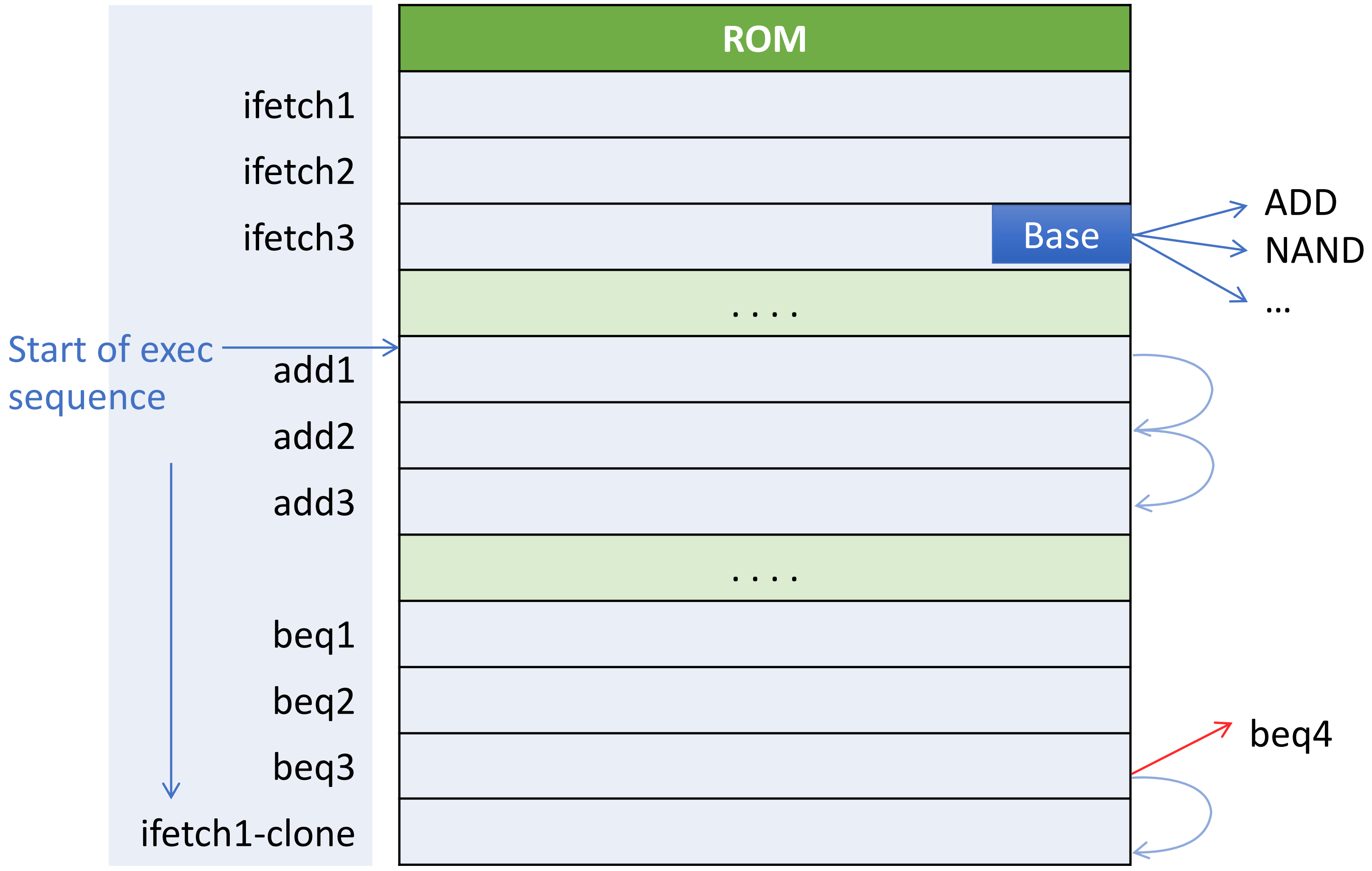
	Drive Signals					Load Signals						Write Signals						
Current State	PC	ALU	Reg	MEM	OFF	PC	A	B	MAR	IR	Z	MEM	REG	func	Reg Sel	**M**	T	Next State
0000000000	1						1		1									00001
0000000001				1						1								00010
0000000010		1				1								11		1		10000

- So how do we make it take that multi-way branch?
- Just set the M bit at 0000000010!

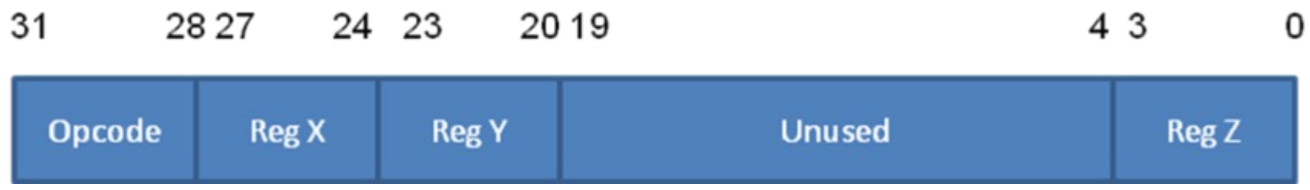
Next State After the Last State of Fetch



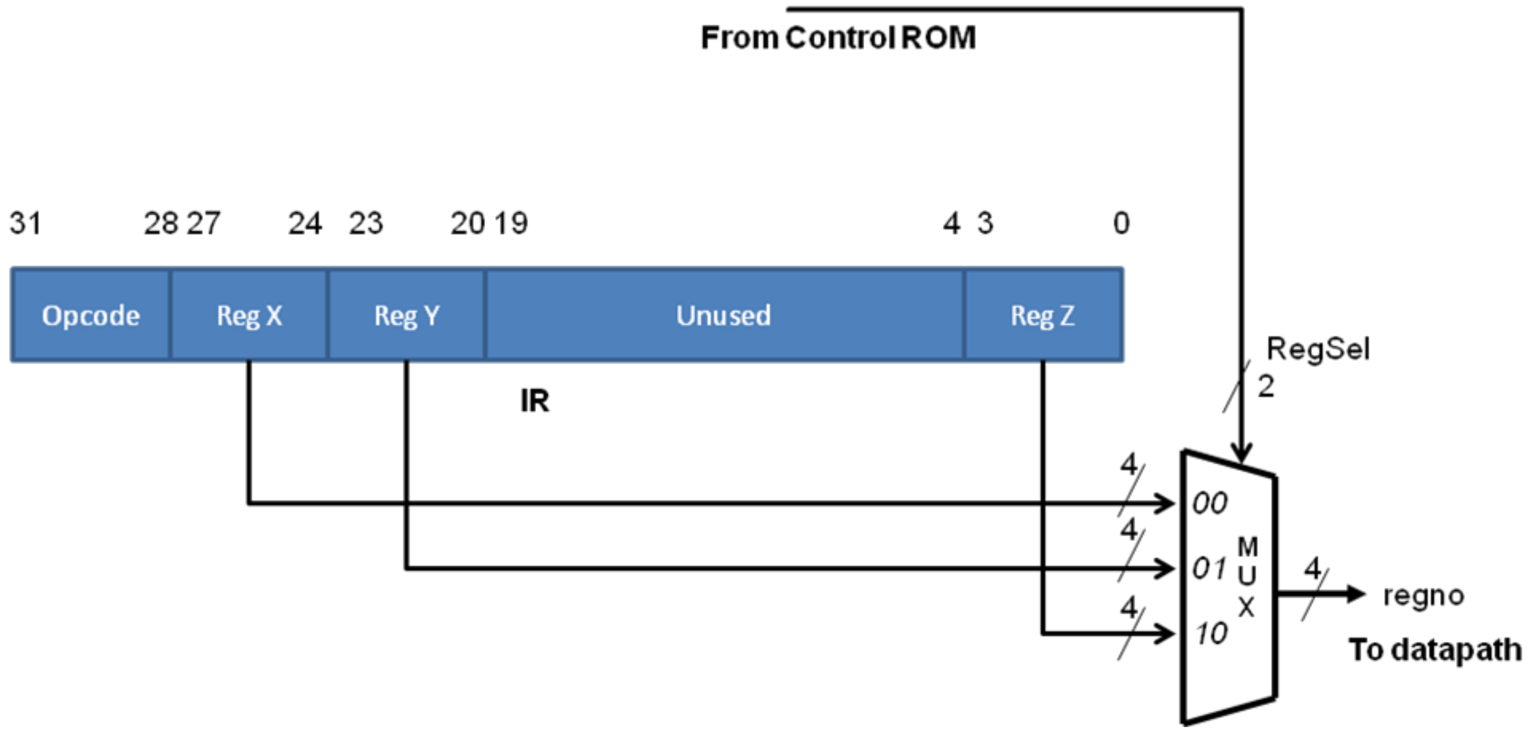
ROM Contents



EXECUTE state: ADD instruction



- $R_X \leftarrow R_Y + R_Z$

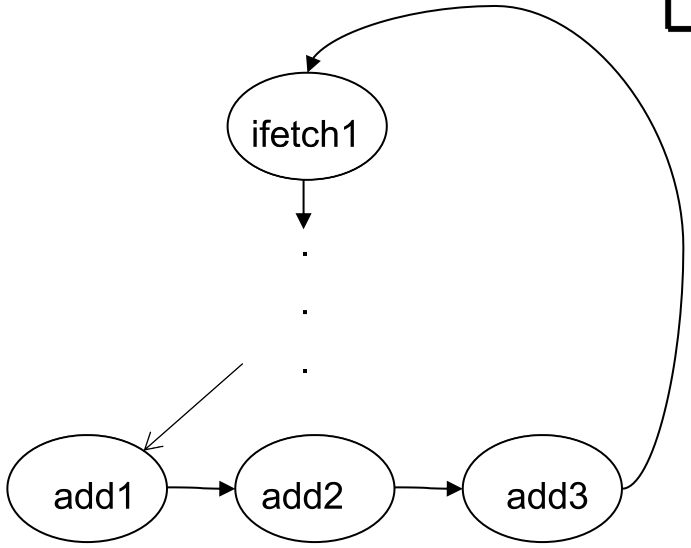


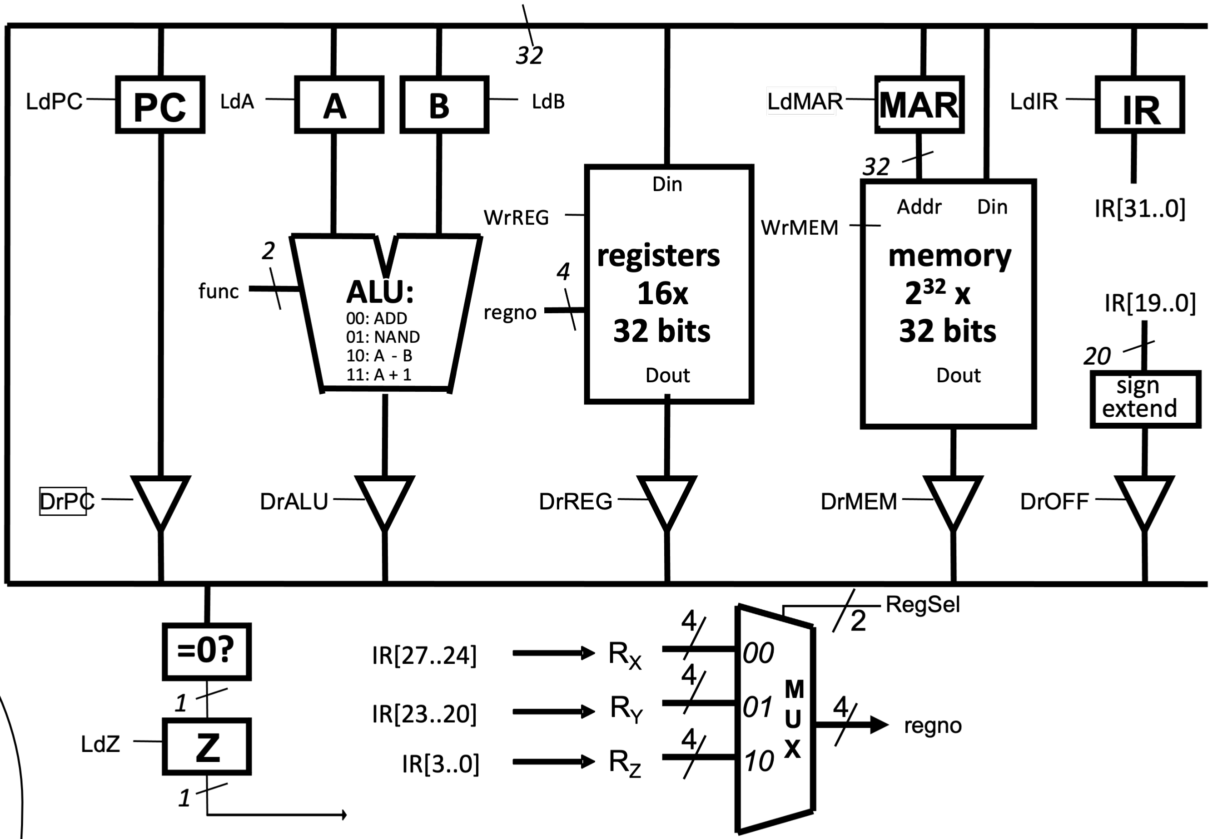
EXECUTE state: ADD instruction

add1
- $R_y \rightarrow A$
- Control signals needed:
  - RegSel = 01
  - DrREG
  - LdA

add2
- $R_z \rightarrow B$
- Control signals needed:
  - RegSel = 10
  - DrREG
  - LdB

add3
- $A+B \rightarrow R_x$
- Control signals needed:
  - func = 00
  - DrALU
  - RegSel = 00
  - WrREG



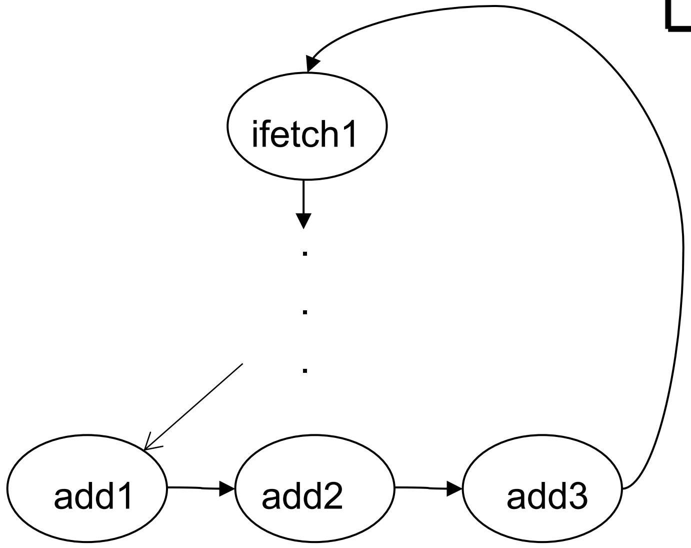


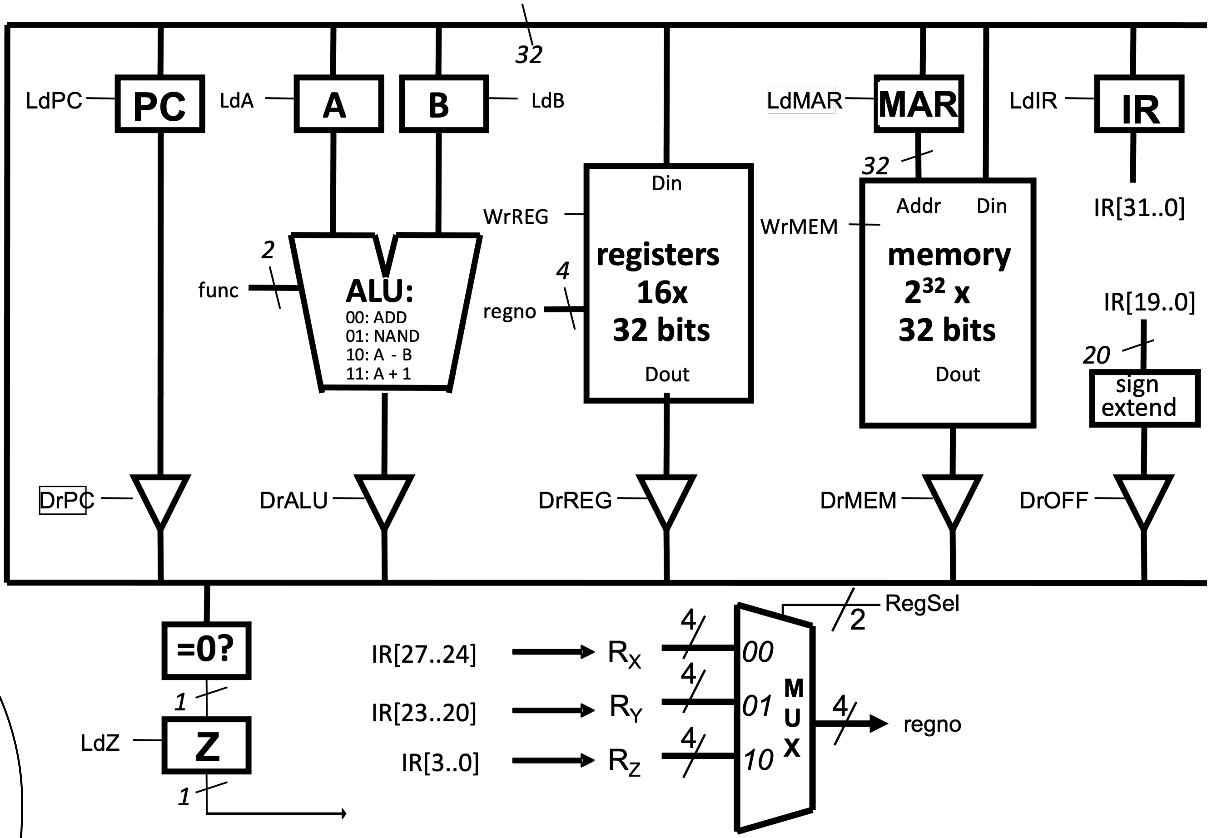
EXECUTE state: ADD instruction

add1
- Ry → A
- Control signals needed:
  - RegSel = 01
  - DrREG
  - LdA

add2
- Rz → B
- Control signals needed:
  - RegSel = 10
  - DrREG
  - LdB

add3
- A+B → Rx
- Control signals needed:
  - func = 00
  - DrALU
  - RegSel = 00
  - WrREG





What must be changed in ADD to implement NAND?

EXECUTE state: JALR instruction

JALR instruction does the following:

$R_Y \leftarrow PC + 1$

$PC \leftarrow R_X$

jalr1

PC → Ry

Control signals needed:

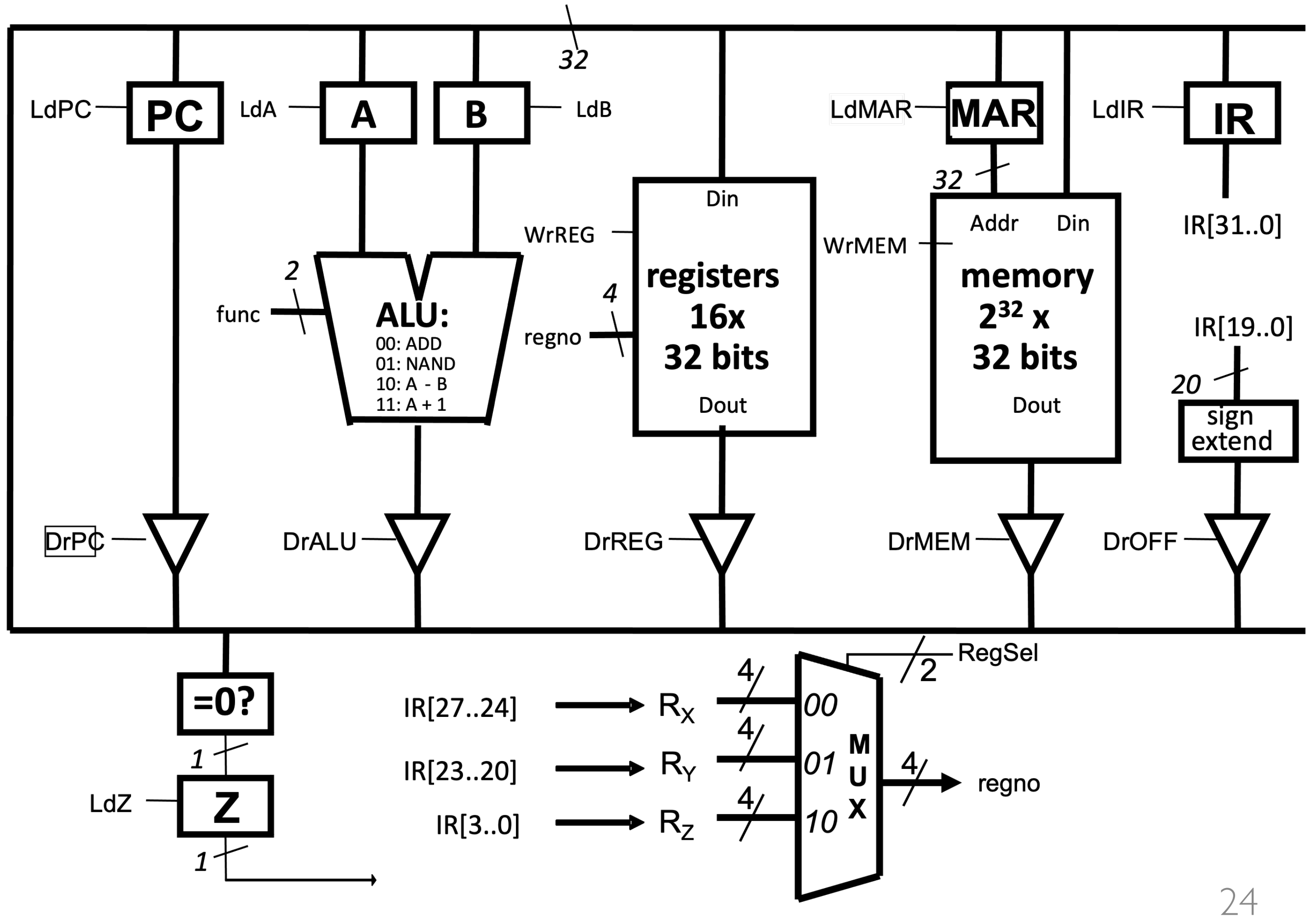
- DrPC
- RegSel = 01
- WrREG

jalr2

Rx → PC

Control signals needed:

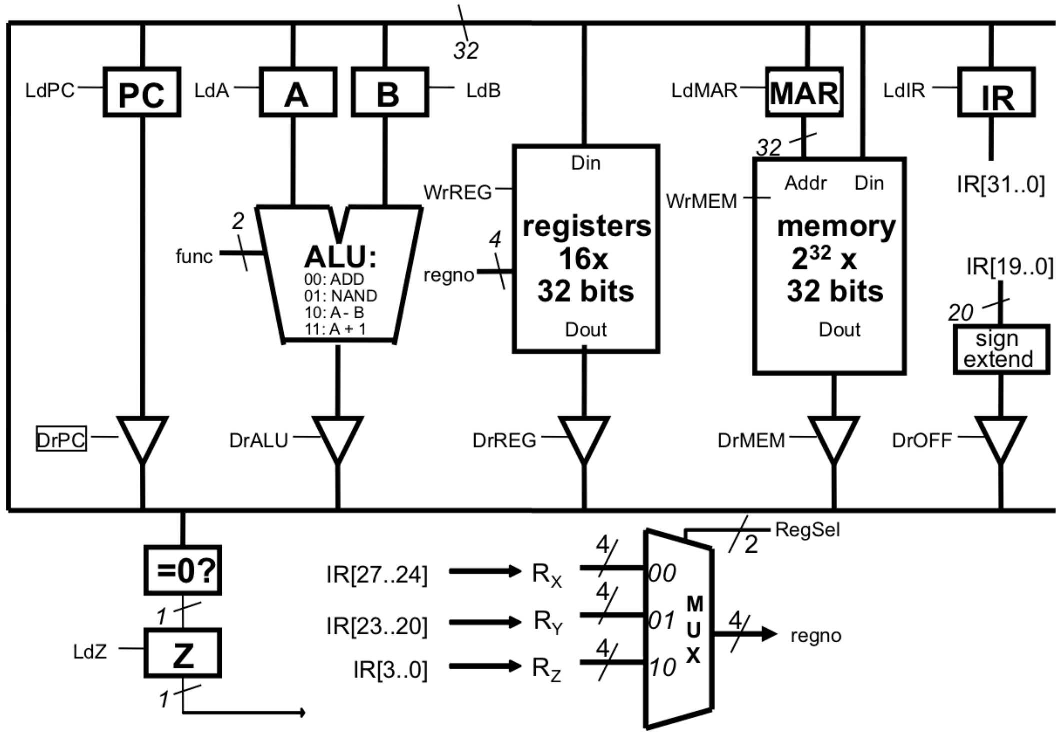
- RegSel = 00
- DrREG
- LdPC



Session ID: dag2200

Question

When all of the control signals are zero in the LC-2200 datapath, what value is being presented by the ALU to DrALU?



0% A. A + B

0% B. The value of one of the registers

0% C. Zero

0% D. Floating

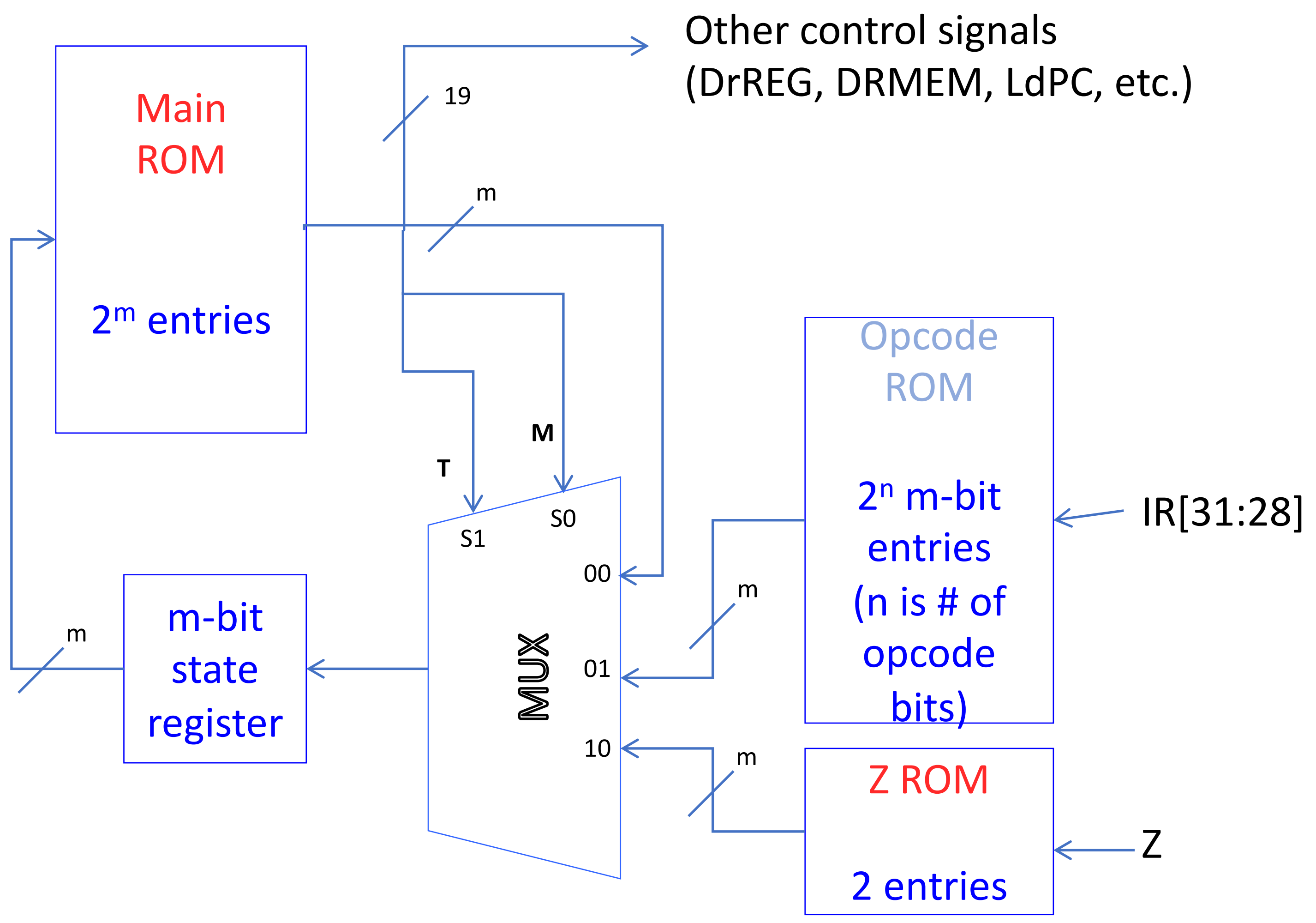
Alternative Style of Control Unit Design

A number of different approaches may be used to implement the Control Unit

Microprogrammed Control

- As presented our design works
- Problem: Too slow
  - Solution: Pre-fetch the next microinstruction
- Problem: Too much memory required
  - Solution: OR the opcode with the next state value 10000 instead of pre-pending it
  - Solution: Use more than one ROM and more sophisticated Decode/BEQ logic
    - One set of ROMs for which state comes next (i.e., "Next State")
    - One for what the control outputs should be in the state

3-ROM Microsequencer



Space/Time Tradeoff

- Flat ROM
  - More space (since we increased the ROM by a factor of 32 for the occasional address modifiers, but have extra ROM space)
  - Faster since only one ROM access in each microinstruction
- Micro sequencer (3-ROM control unit)
  - Less space (main ROM much smaller than Flat ROM)
  - Slower since additional ROM access in every clock cycle

Hardwired Control

- State machine can be represented as sequential logic truth table
- Thus can be implemented using normal combinational logic or FPGA

- Can produce boolean function for each control signal
  - E.g., DrPC = ifetch1 + jalr1 + beq4 + …

Control Regime	Pros	Cons	Comment	When to use	Examples
Micro-programmed	Simplicity, maintainability, flexibility Rapid prototyping	Potential for space and time inefficiency	Space inefficiency may be mitigated with vertical microcode Time inefficiency may be mitigated with prefetching	For complex instructions, and for quick non-pipelined prototyping of architectures	PDP 11 series, IBM 360 and 370 series, Motorola 68000, complex instructions in Intel x86 architecture
Hardwired	Amenable for pipelined implementation Potential for higher performance	Potentially harder to change the design Longer design time	Maintainability can be increased with the use of structured hardware such as PLAs and FPGAs	For High performance pipelined implementation of architectures	Most modern processors including Intel Pentium series, IBM PowerPC, MIPS

Interrupts, Traps and Exceptions

- Interrupts, traps and exceptions are discontinuities in program flow
- Students asking a teacher questions in a classroom is a good analogy to the handling of discontinuities in program flow

Discontinuities in program execution

We must first understand

- Synchronous events: Occur at well defined points aligned with activity of the system
  - Making a phone call
  - Opening a file

- Asynchronous events: Occur unexpectedly with respect to ongoing activity of the system
  - Receiving a phone call
  - A user presses a key on a keyboard

Discontinuities in program execution

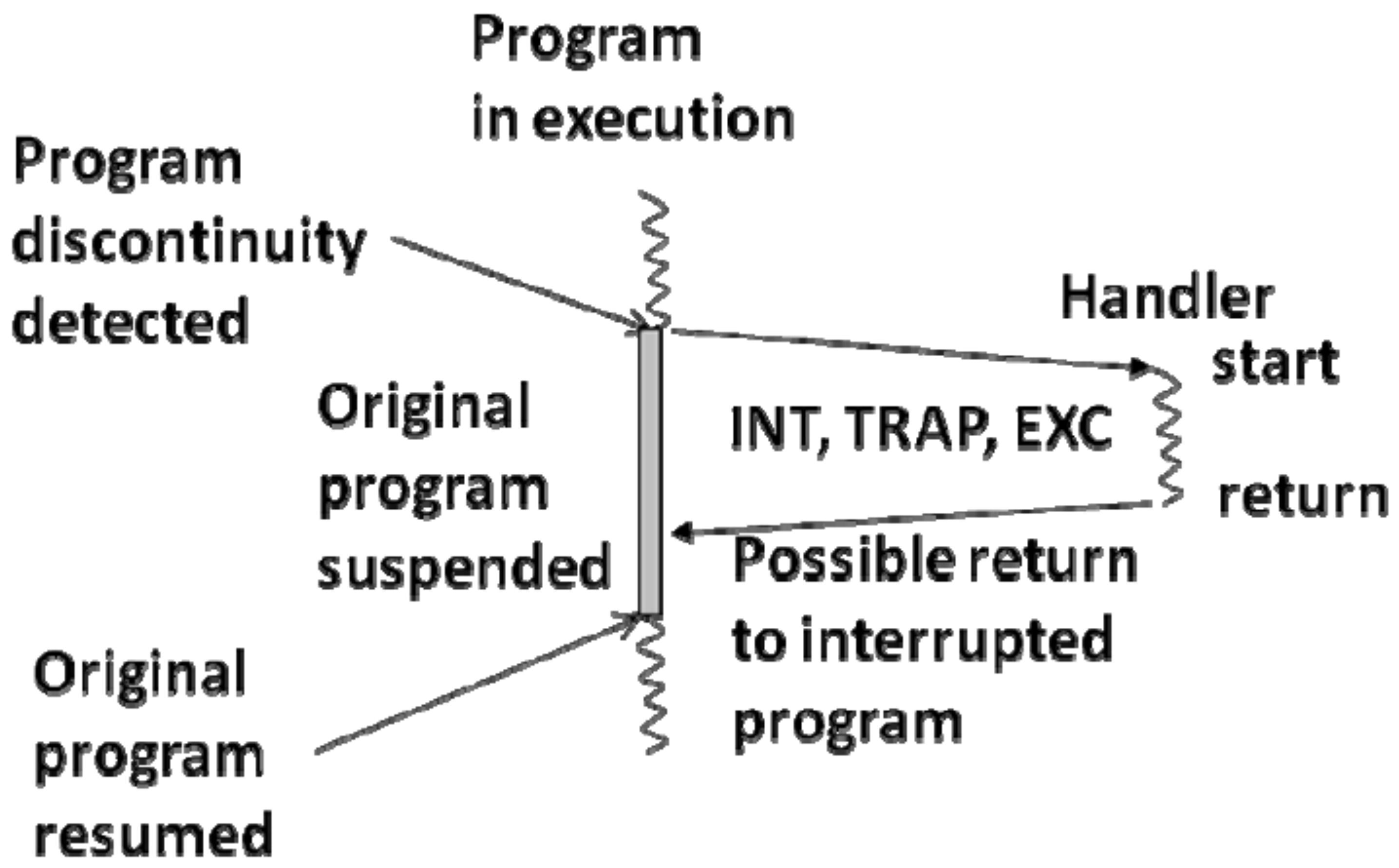
Definitions

- **Interrupts**: Asynchronous events usually produced by I/O devices which must be handled by the processor by interrupting execution of the currently running process
- **Traps**: Synchronous events produced by special instructions typically used to allow secure entry into operating system code
- **Exceptions**: Synchronous events usually associated with software requesting something the hardware can't perform i.e. illegal addressing, illegal op code, etc.

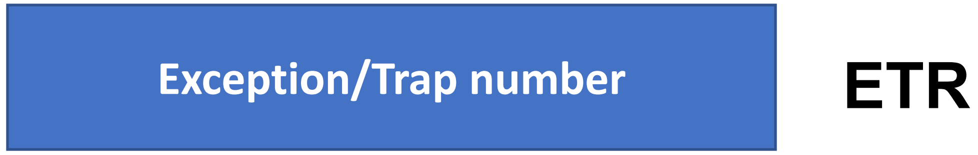
Discontinuities in program execution

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

Execution path

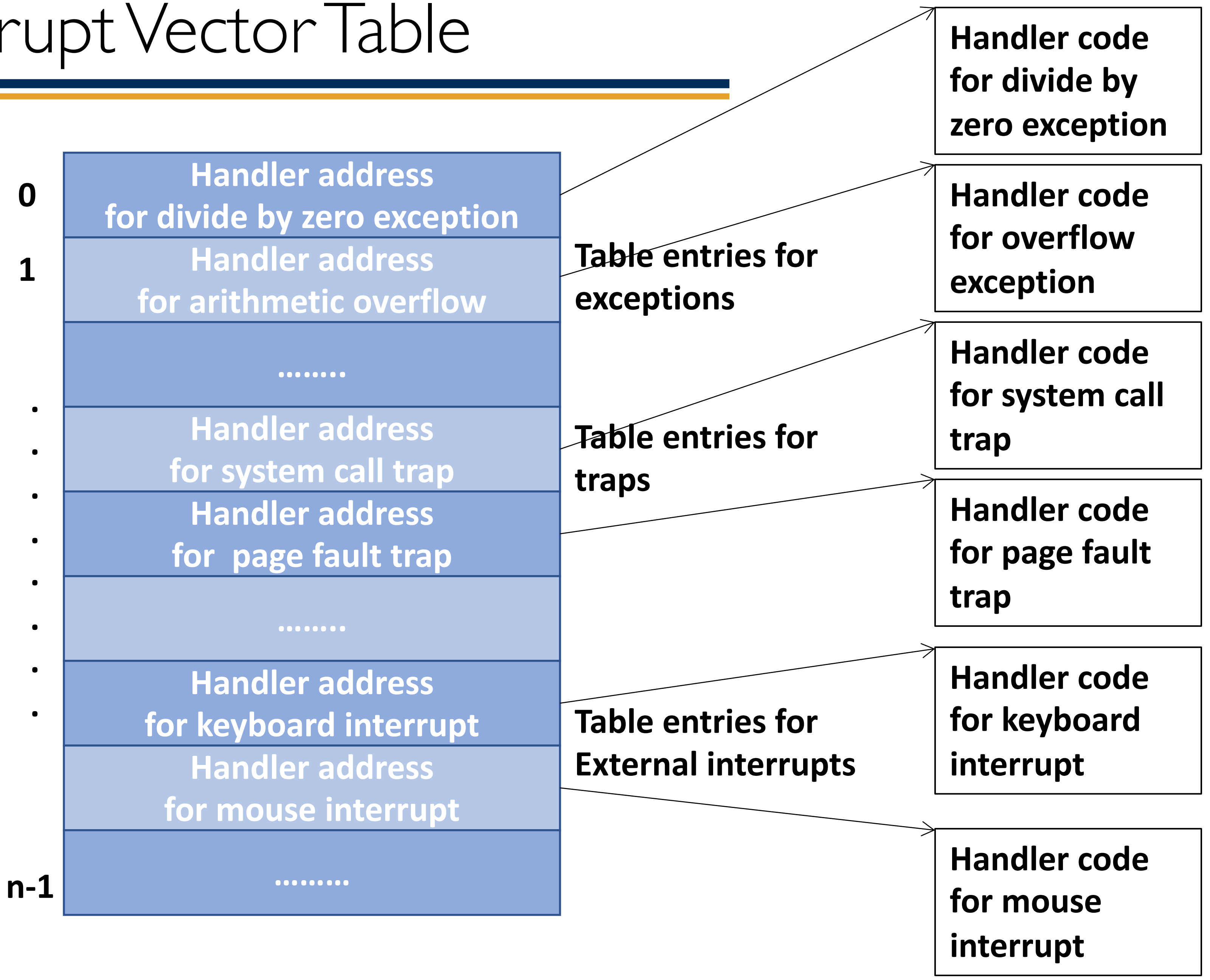


New internal processor register



Will contain a unique number stashed by the hardware to indicate the type of discontinuity

Interrupt Vector Table


0
1
.
.
.
.
.
.
.
.
n-1
Handler address for divide by zero exception
Handler address for arithmetic overflow
....
Handler address for system call trap
Handler address for page fault trap
....
Handler address for keyboard interrupt
Handler address for mouse interrupt
....
Table entries for exceptions
Table entries for traps
Table entries for External interrupts
Handler code for divide by zero exception
Handler code for overflow exception
Handler code for system call trap
Handler code for page fault trap
Handler code for keyboard interrupt
Handler code for mouse interrupt

Dealing with program discontinuities

…some similarities with a function call, but several differences too

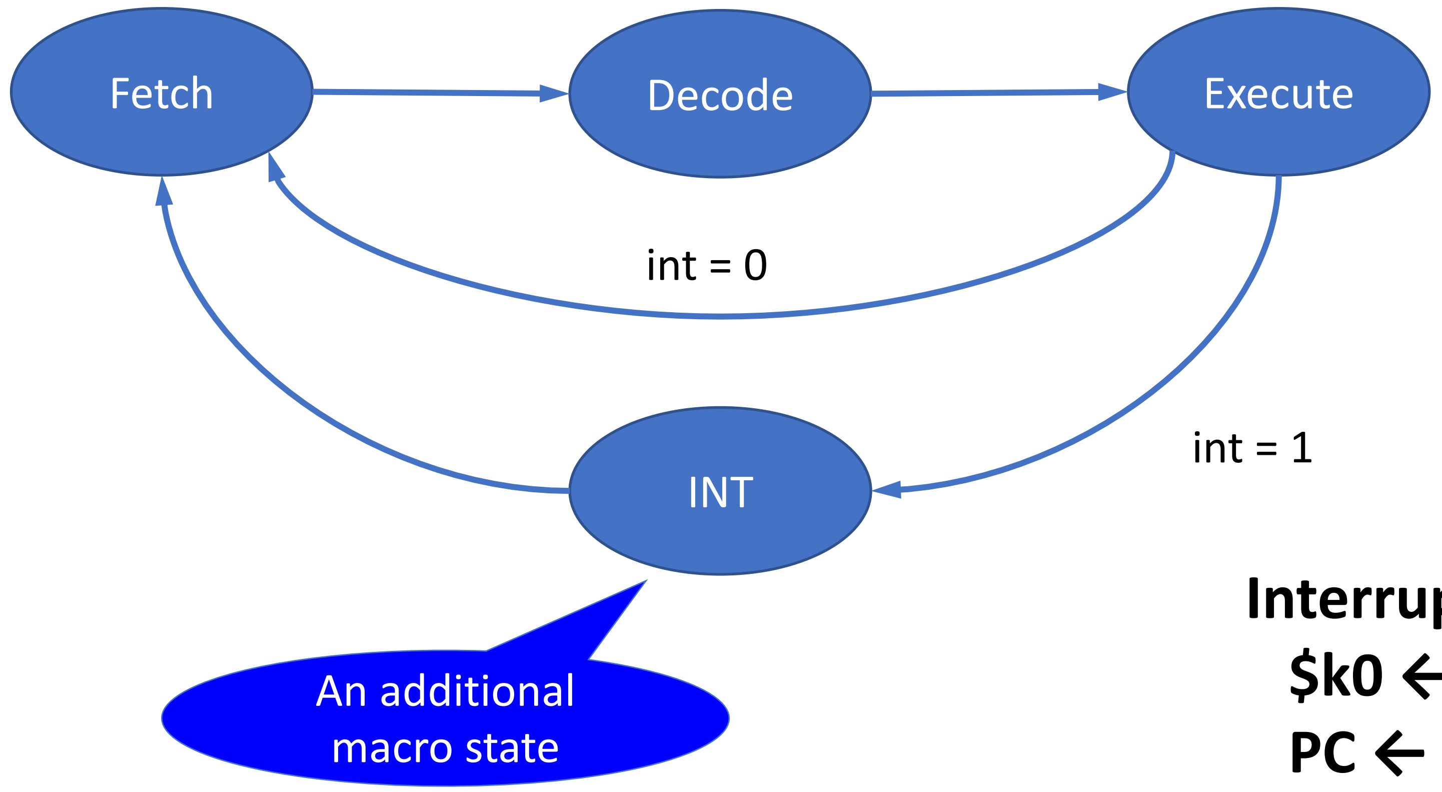
- Can happen anywhere even in the middle of an instruction execution.
- Unplanned for and forced by the hardware. Hardware has to save the program counter since we are jumping to the handler.
- Address of the handler is unknown. Therefore, hardware must manufacture an address.
- Since hardware saved the PC, handler has to discover where to return upon completion.

Architectural enhancements to handle program discontinuities

- When should the processor handle an interrupt?
- How does the processor know there is an interrupt?
- How do we save the return address?
- How do we manufacture the handler address?
- How do we handle multiple cascaded interrupts?
- How do we return from the interrupt?

Modifications to FSM

Where should we take an interrupt?



Interrupt:

$k0 ← PC

PC ← new PC

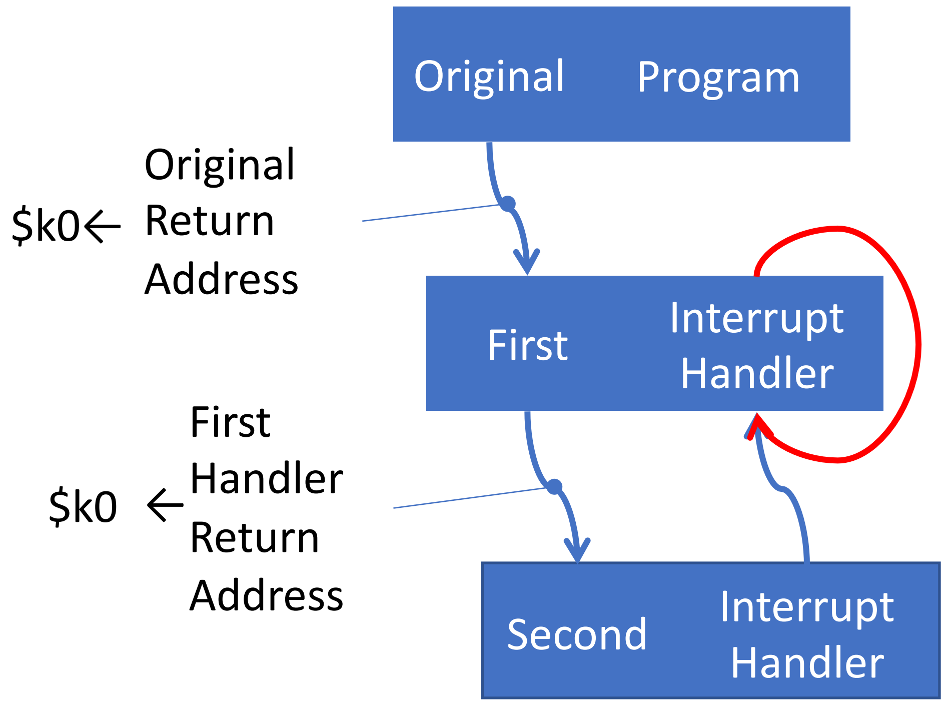
What needs to happen in software?

```
Handler:
  save processor registers;
  execute device code;
  restore processor registers;
  return to original program;
```

That's great, but…

- There are a couple of rubs.
- What happens when an interrupt handler takes an interrupt?

Handling cascaded interrupts



What needs to happen …

```
Handler:
  save processor registers
      (including $k0);
  execute device code;
  restore processor registers
      (including $k0);
  return to original program;
```

That's great, but…

- There are a couple of rubs.
- What happens when an interrupt handler takes an interrupt?
- OK. That's better. Save/restore $k0 in the handler.
- But one more little thing…
- What happens if the second interrupt hits *before* we save $k0?

What needs to happen …

```
Handler:
  save processor registers
     (including $k0);
  execute device code;
  restore processor registers
     (including $k0);
  return to original program;
```

What if an interrupt happens here?

No problem. We'll save $k0 first

How many instructions does it take to push a register on the stack?

What needs to happen …

```
Handler:
  save processor registers
      (including $k0);
  execute device code;
  restore processor registers
      (including $k0);
  return to original program;
```

Store $k0 first.

It takes
•Decrement $sp
•Store $k0,0($sp)

What if the interrupt happens after the Decrement!?!?

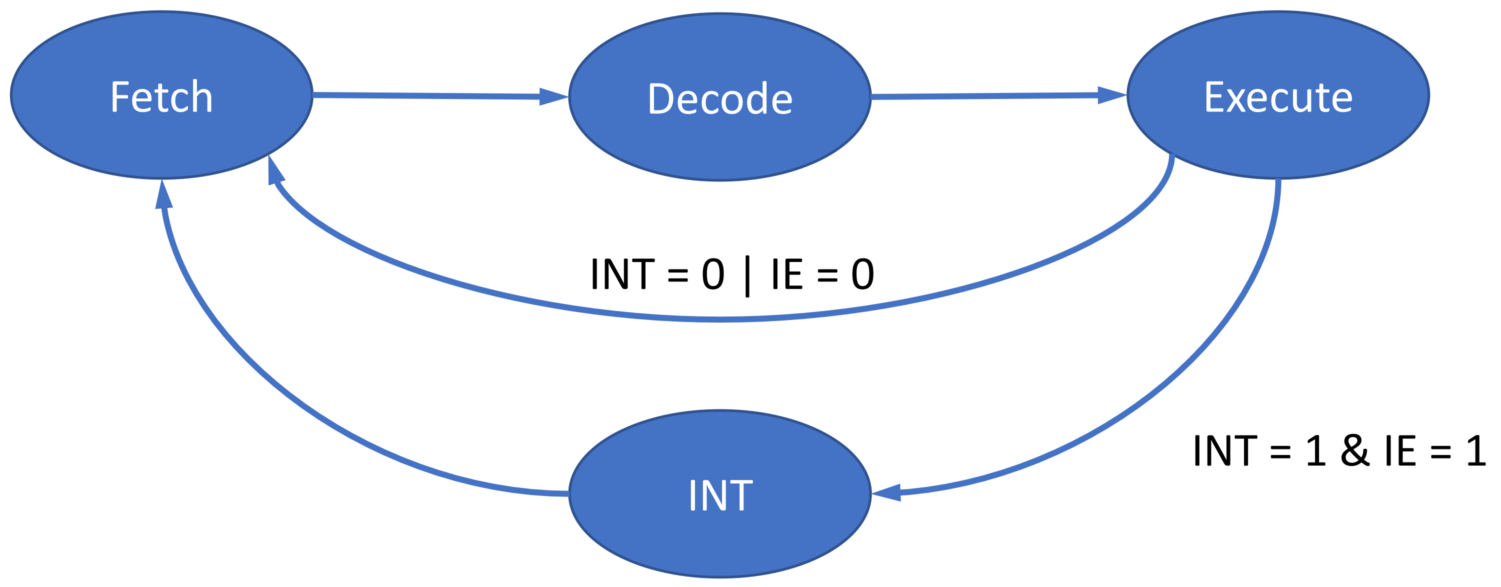
What are we lacking?

- We don't have a way to prevent an interrupt from happening between certain instructions
- In other words, we need for groups of machine instructions to behave **atomically** – i.e. as if they all were executed as a single instruction
- How could we do that?
- We could turn off interrupts between instructions?

The plan

- Create a new processor register, IE, that is 1 when interrupts are enabled
- For an interrupt to be recognized, i.e. for the microcode to advance to the INT macro state, an interrupt must be asserted **and** IE must be 1
- In the INT macro state, turn off IE before fetching the first instruction in the handler
- We need two more instructions: EI and DI to respectively set IE to 1 and 0.
- Use EI after pushing $k0 on the stack

Handling cascaded interrupts



Interrupt:
disable Ints
$k0 ← PC
PC ← new PC

Add 2 new instructions
Enable Ints (EI)
Disable Ints (DI)

Yay! This will work perfectly!

```
Handler:
  save $k0;
  enable interrupts
  save processor registers;
  execute device code;
  restore processor registers;
  disable interrupts;
  restore $k0;
  enable interrupts;
  return to original program;
```

Or does it?

What if an interrupt occurs here?

Returning from the handler

- Returning involves jumping to the address in $k0 which can be accomplished with

```
jalr $k0, $zero
```

- But as we have just seen, an interrupt at precisely the wrong moment would destroy $k0 and cause a failure
- What do we need?
- All this needs to be atomic, too!

```
restore $k0;
enable interrupts;
return to original program;
```

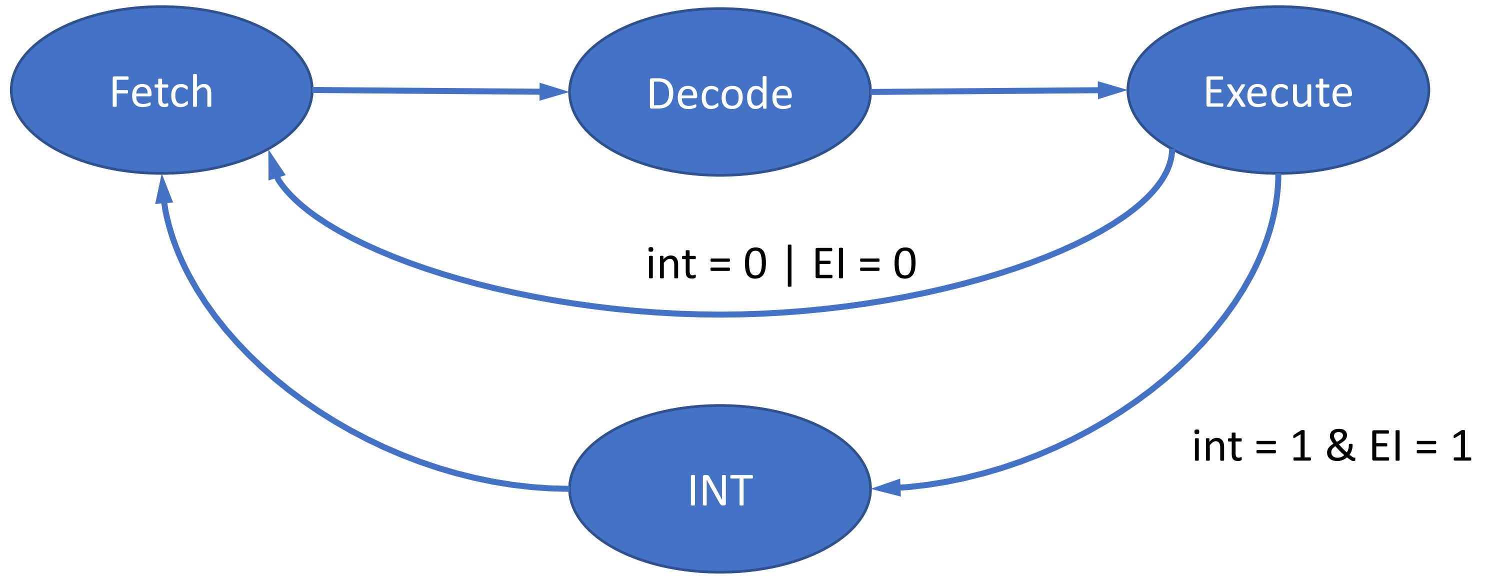
Returning from the handler

- So we need another new instruction, RETI
- It atomically enables interrupts and sets the PC to return from the handler

- RETI:

```
PC ← $k0
EI ← 1
```

Handling cascaded interrupts



Interrupt:
Disable Ints
$k0 ← PC
PC ← new PC

Add 3 new instructions
Enable Ints (EI)
Disable Ints (DI)
Return from interrupt (RETI)

Summary of architectural enhancements to LC-2200 to handle interrupts (so far)

- Three new instructions to LC-2200:
    - Enable interrupts (EI)
    - Disable interrupts (DI)
    - Return from interrupt (RETI)

- 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

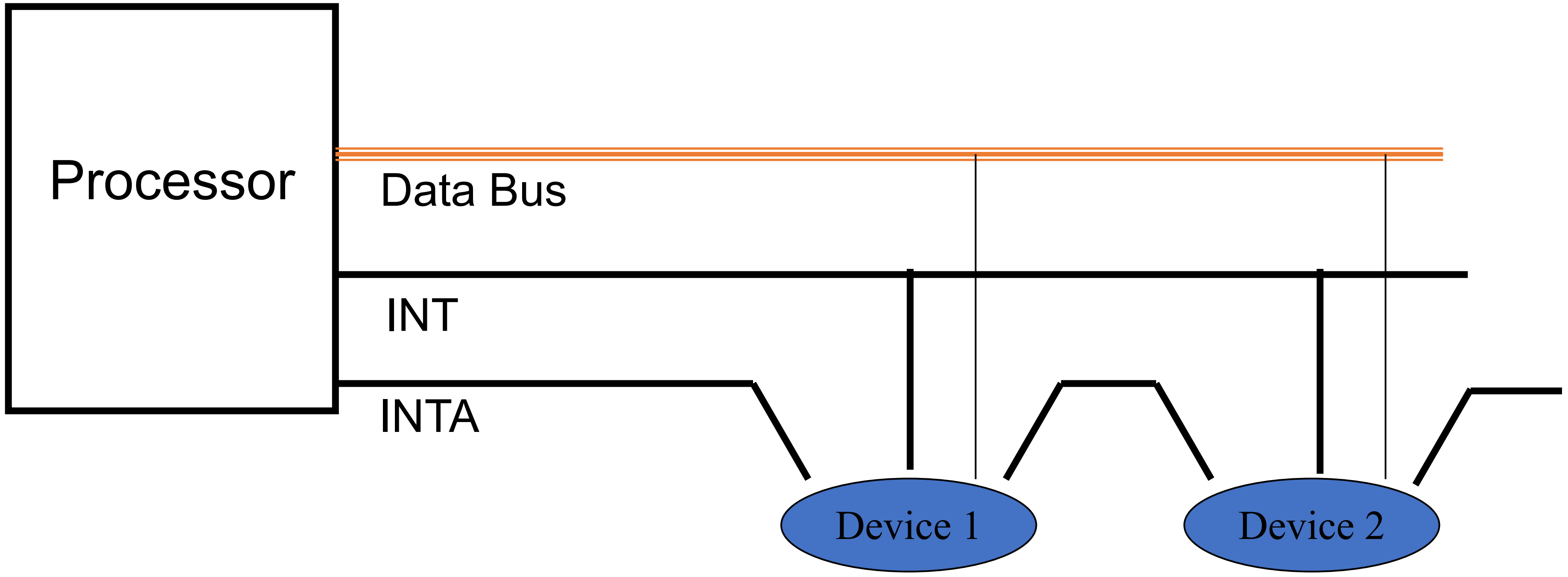
- Upon returning from an interrupt (RETI), store $k0 into the PC and enable interrupts.

Hardware details for handling external interrupts

- What we have presented thus far is what is required for interrupts, traps and exceptions

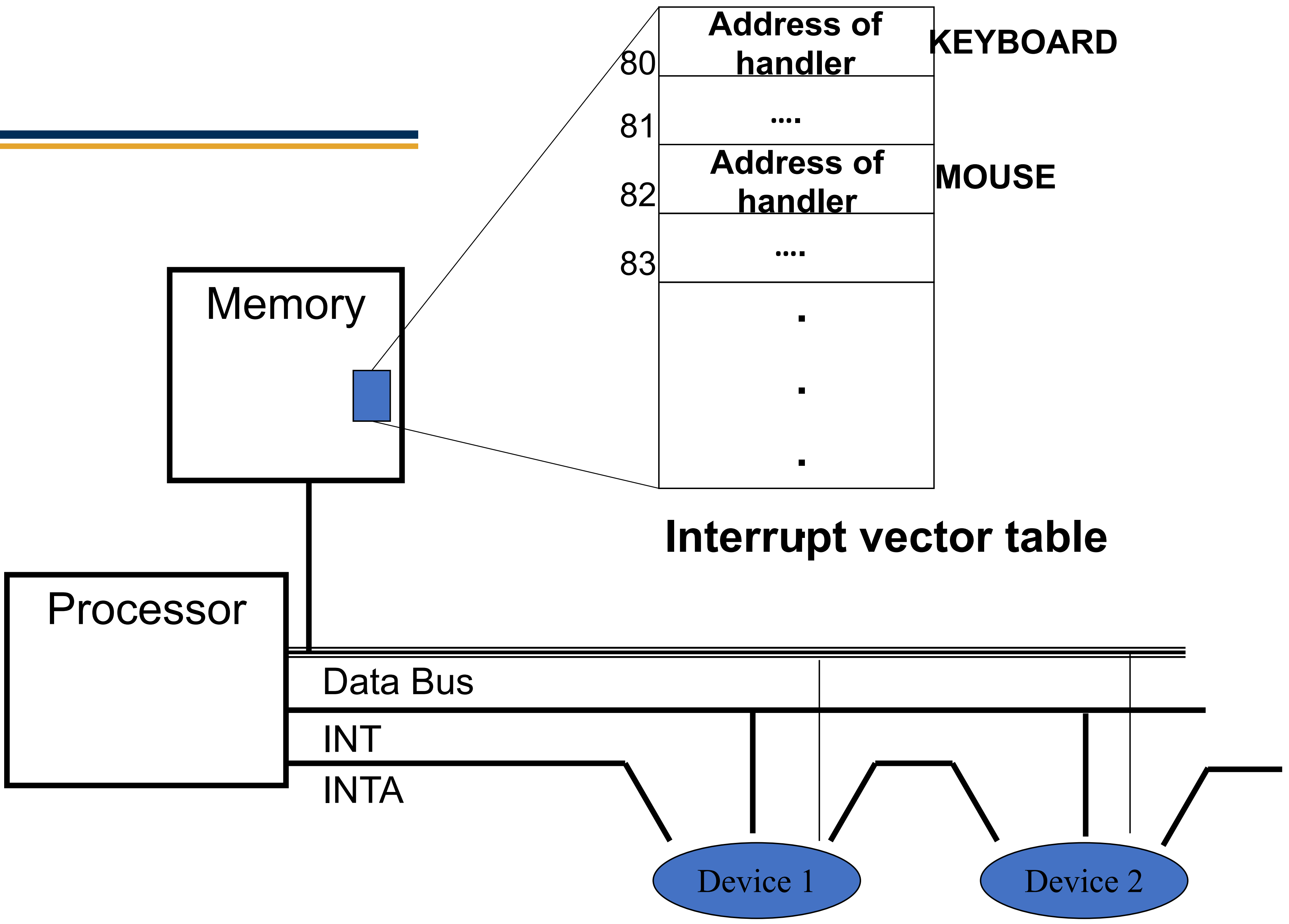
- What do we need specifically for external interrupts?
  - How does the processor know an external interrupt occurred?

Wiring for external interrupts



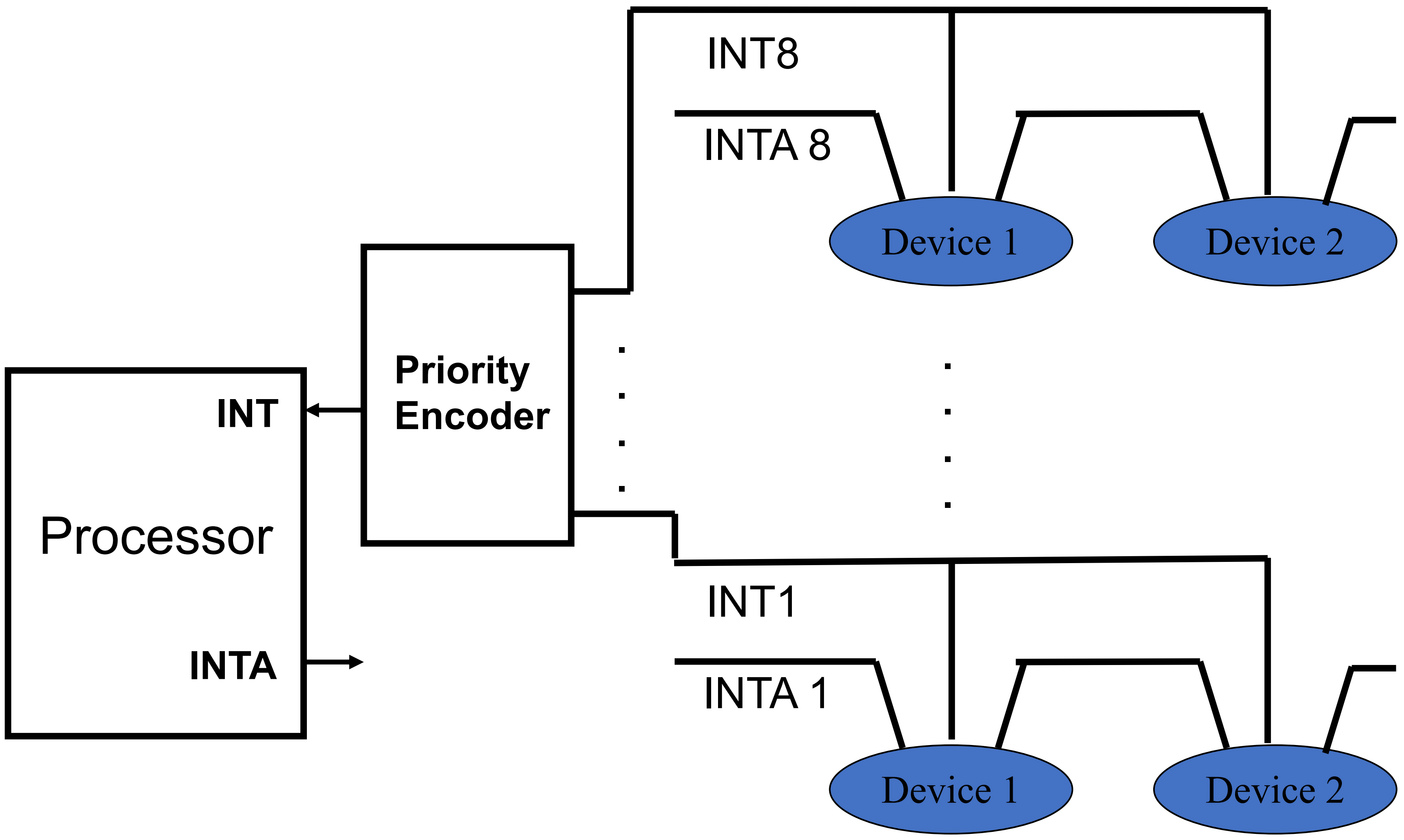
What happens at an interrupt?

- Device asserts the INT bus (it's wired so multiple devices can do this simultaneously)
- At the completion of the current instruction, CPU sees INT signal (IE = 1 & INT = 1) and microcode cycles into the INT macro state
- Microcode raises the INTA signal line
- Devices pass-through the INTA signal if they are not interrupting; otherwise the first interrupting device asserts its ID on the data bus
- Microcode reads the data bus and uses the ID as an index to determine which entry in the IVT to use to set the PC



Interrupt vector table

Multiple interrupt priority levels



Priority Encoder (PE)

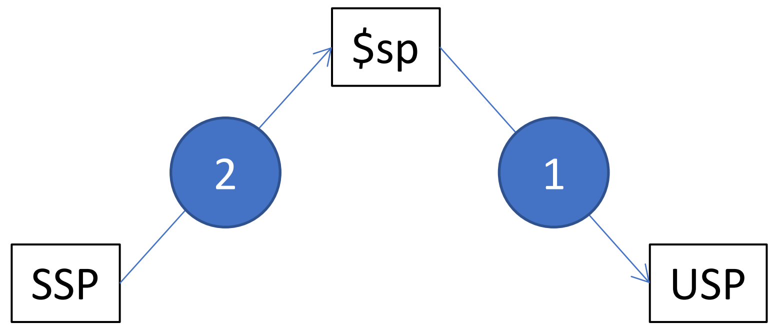
- A priority encoder takes 2^n inputs and produces a 1-bit INT output and an n-bit ID output.
- If any of the input lines is high, the PE asserts the INT output
- The PE asserts the encoded value of the first high input line onto the ID output
  - E.g. if input 5 and 7 are high on a 3-bit PE, then it asserts INT and ID=101
  - If only input 7 is high, then it asserts INT and ID=111

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?

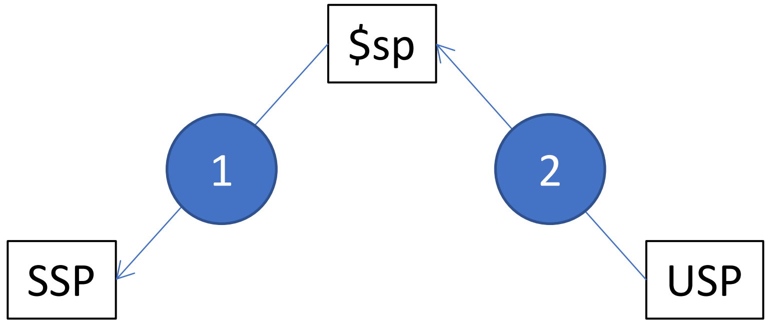
Stack for saving/restoring

- 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

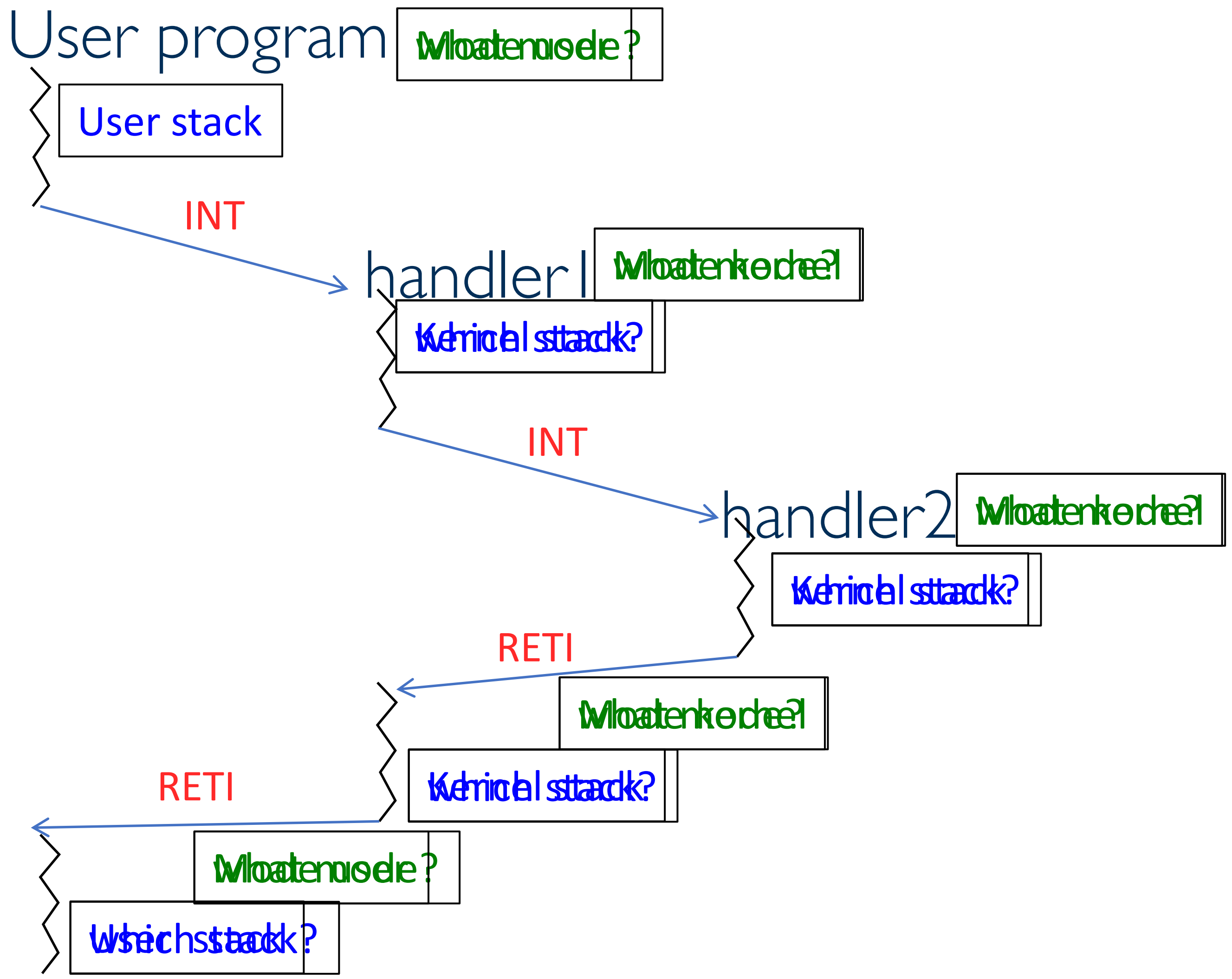


Stack for saving/restoring

- Use system stack for saving all necessary information
- Upon completion of interrupt restore registers, etc.
- Then 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 ← PC
    Assert INTA to acknowledge interrupt
    Receive IV (interrupt vector) from the device on the data bus
    PC ← Mem[IV]
    if user mode,
        USP ← $sp; $sp ← SSP
    Push mode on stack
    mode ← kernel
    Disable interrupts
```

RETI instruction:

```
    PC ← $k0
    Pop mode from system stack
    if user mode,
        SSP ← $sp; $sp ← USP
    Enable interrupts
```

A working interrupt handler

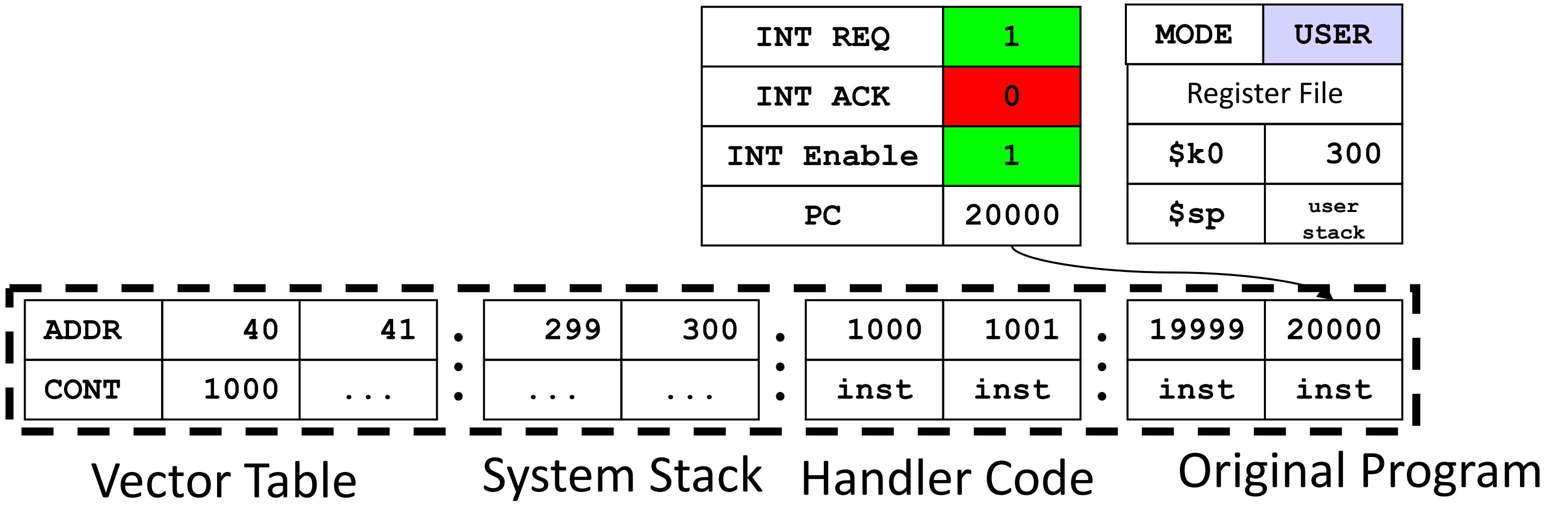
```
Handler:
  // handler starts with interrupts disabled
  push $k0 onto system stack;
  enable interrupts;

  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

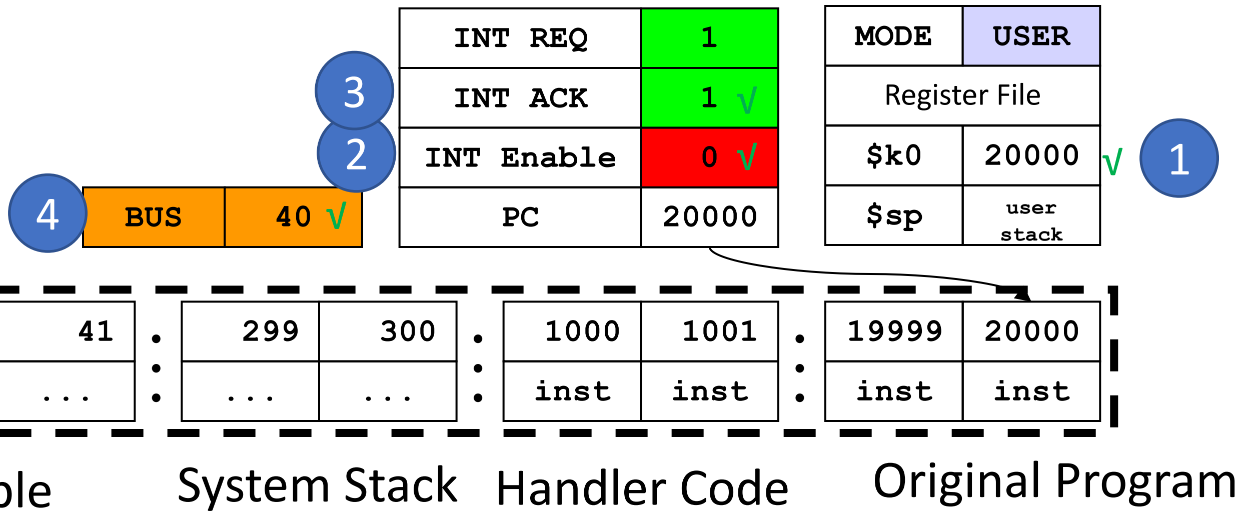
Executing instruction at 19999. The PC has already been incremented. Device signals interrupt in middle of instruction. $sp points to user stack



Putting it all together

Interrupt has been noticed.

1. $k0 gets PC.
2. Interrupts are disabled.
3. Interrupt is acknowledged.
4. Device puts vector on bus.



Putting it all together

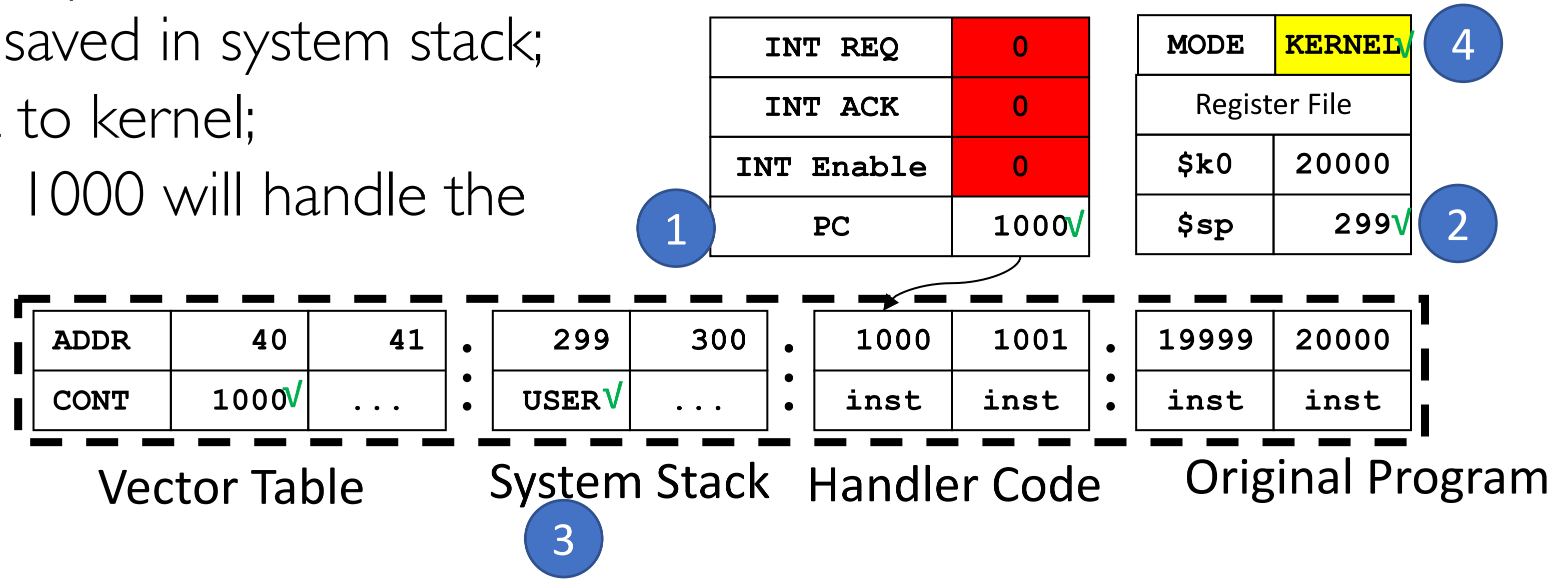
1 Handler address is put into PC

2 $sp now points to system stack;

3 Current mode is saved in system stack;

4 New mode is set to kernel;

Interrupt code at 1000 will handle the interrupt. 1



Putting it all together

Handler completes.

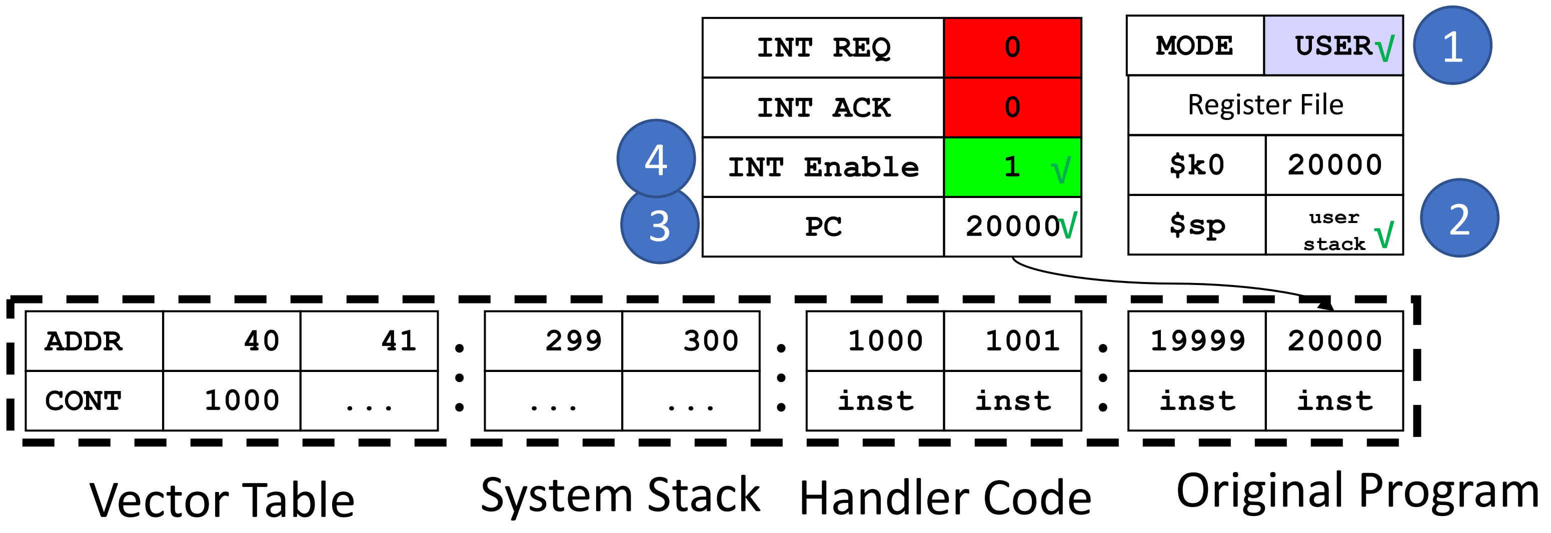
1 RETI instruction restores mode from system stack;

since returning to user program in this example, sets Mode to User;

2 $sp now points to user stack;

3 copies $k0 into PC;

4 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 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.