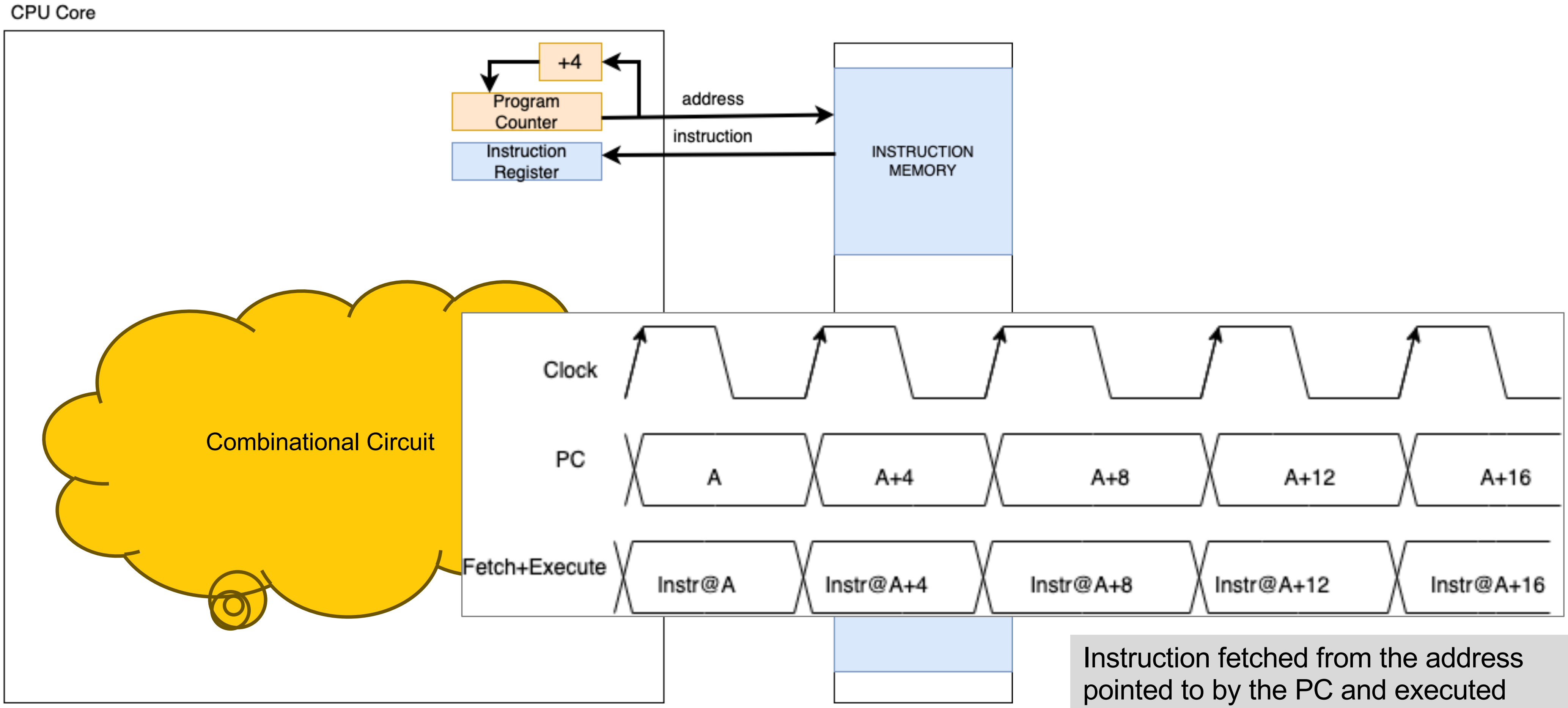
Building a Single Cycle RISC V Processor

Chester Rebeiro
Indian Institute of Technology Madras

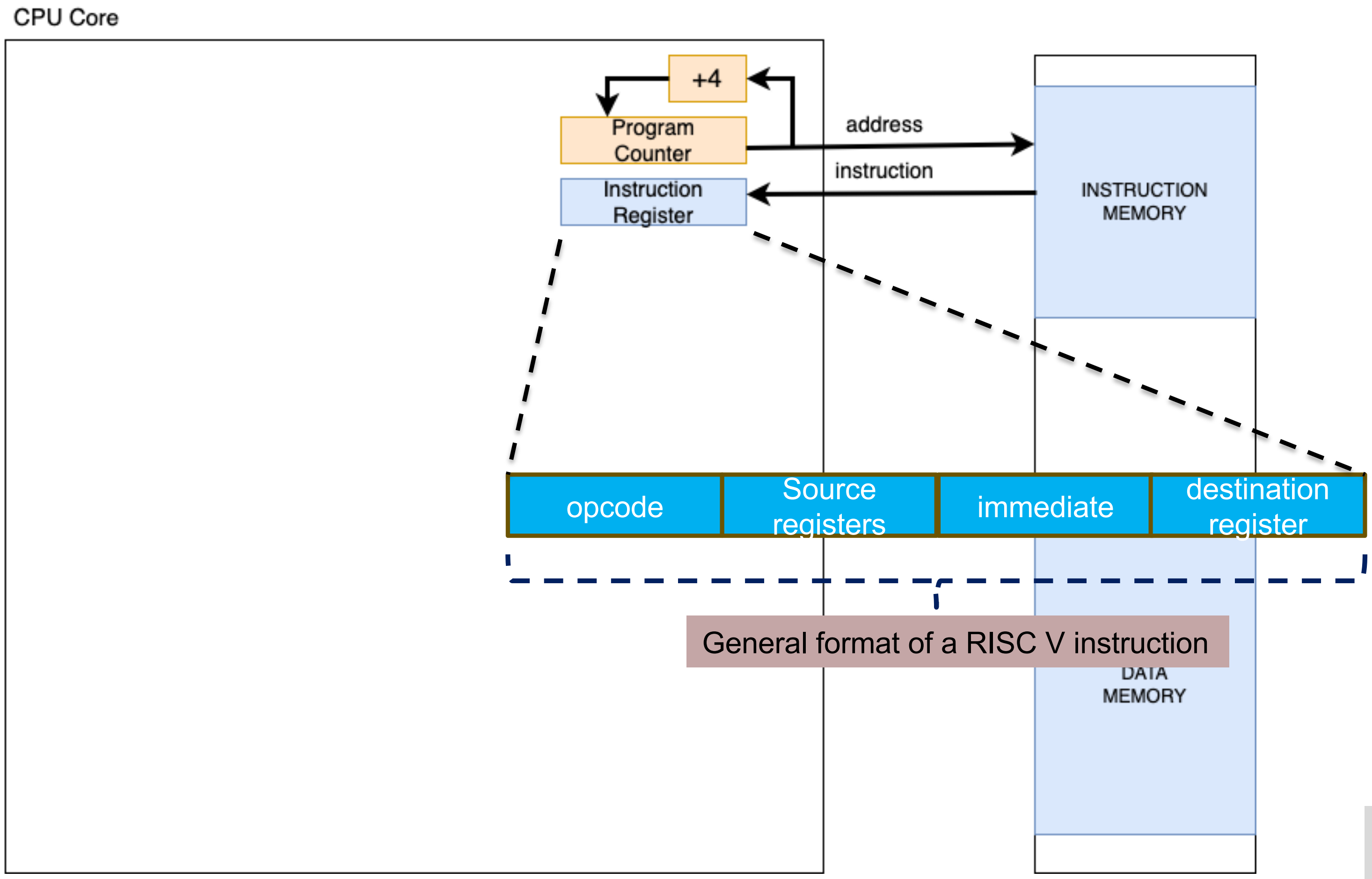
chester@cse.iitm.ac.in

Program Execution. Fetching the Instruction



Instruction fetched from the address pointed to by the PC and executed every rising edge of the clock cycle

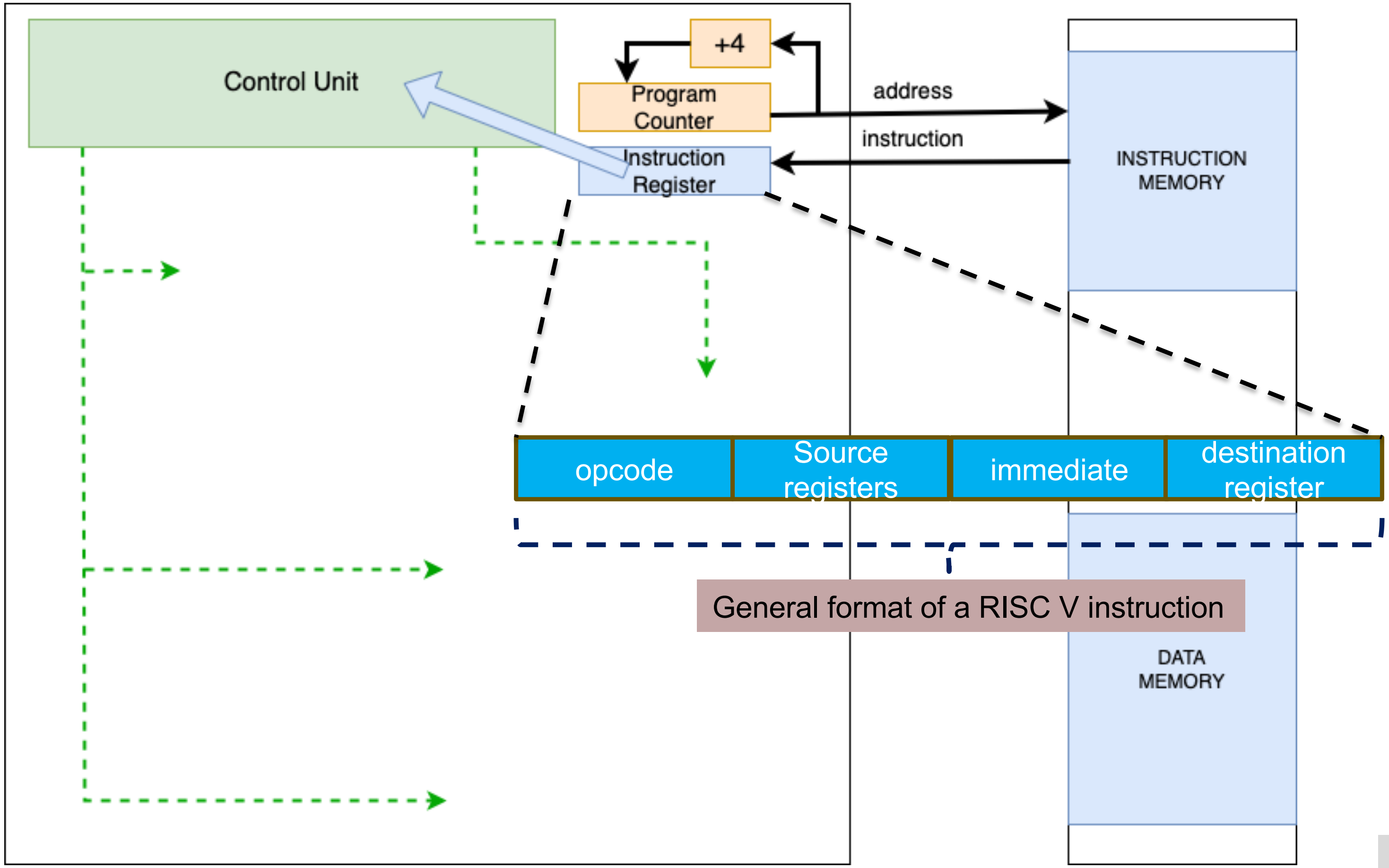
Program Execution. Fetching the Instruction



General format of a RISC V instruction

Instruction fetched from the address pointed to by the PC

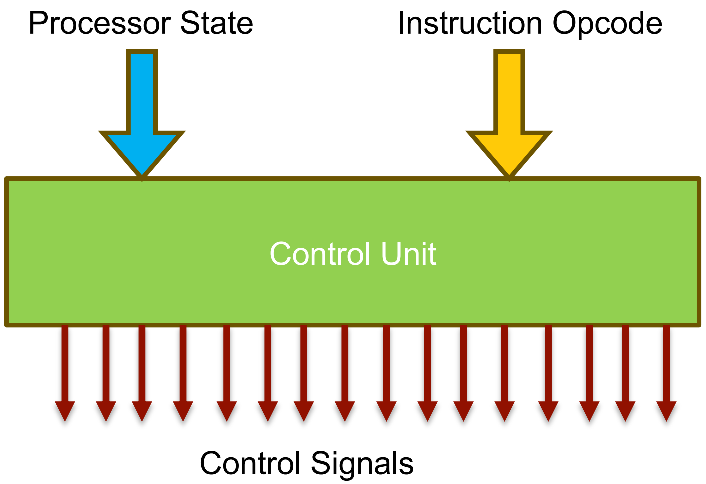
Control Unit



General format of a RISC V instruction

Control signals generated
Based on instruction opcode and functions

Control Unit



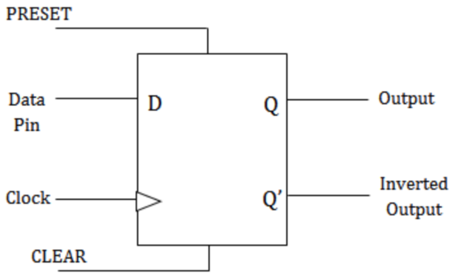
Truth Table

<table>
<tr><th colspan="10">Processor State Signals</th><th colspan="10">Instruction Opcode</th><th colspan="12">Control Signals</th></tr>
<tr><td>0</td><td>0</td><td>1</td><td>x</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>x</td><td>0</td><td>x</td><td>0</td><td>0</td><td>0</td><td>x</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>x</td><td>0</td><td>0</td><td>x</td><td>x</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>x</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>x</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>x</td><td>1</td><td>x</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>x</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td></tr>
<tr><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td></tr>
<tr><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td></tr>
<tr><td>x</td><td>1</td><td>1</td><td>x</td><td>0</td><td>1</td><td>X</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>X</td><td>0</td><td>0</td><td>1</td><td>0</td><td>X</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

Logic Synthesis

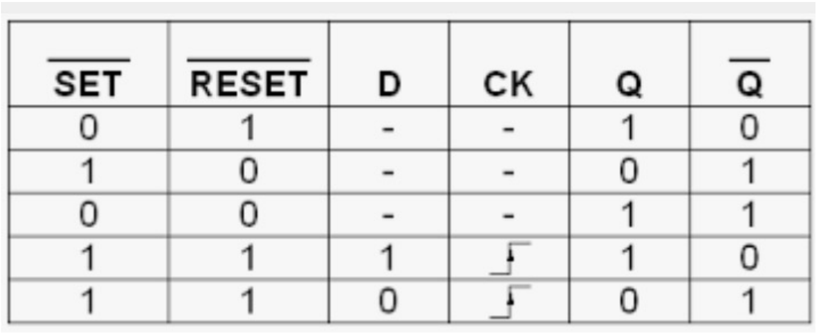
Shows an example of how the control unit can be designed based on truth tables.
In practice, we use Verilog/VHDL to describe the relationships

D Flip Flop and Registers



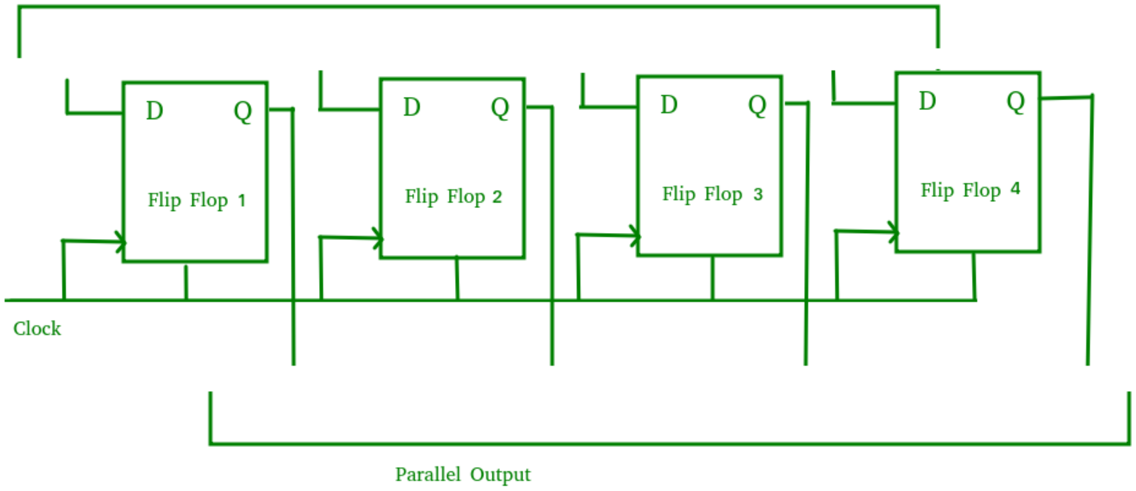
Symbol: D Flip-flop

Truth Table



$\overline{\text{SET}}$	$\overline{\text{RESET}}$	D	CK	Q	$\overline{\text{Q}}$
0	1	-	-	1	0
1	0	-	-	0	1
0	0	-	-	1	1
1	1	1	↑	1	0
1	1	0	↑	0	1

4-bit Register: Parallel In Parallel Out Configuration



A D Flip Flop stores a single bit, forming the basis for a register. A single register, could have 32 or 64 D flip flops, depending on the architecture.

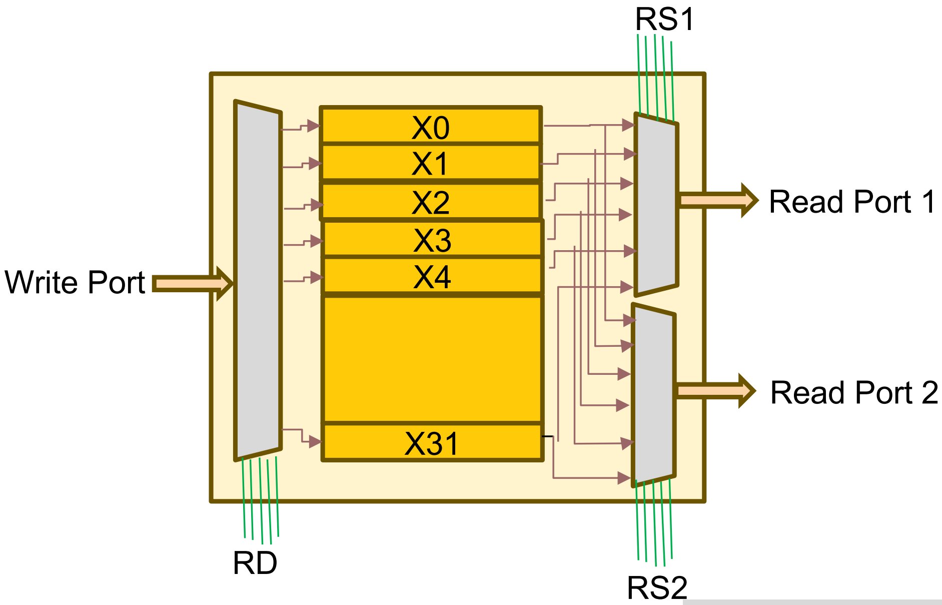
Register Bank

32 registers stored in register file

5-bits required to address registers.

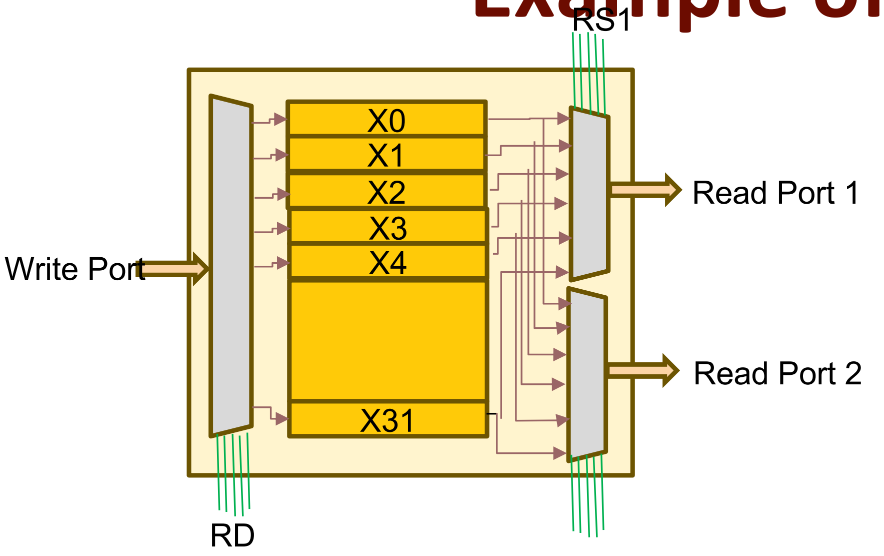
Two source registers and one destination register → therefore dual ported register file
supporting two reads and one write

32 registers stored in register file.
5-bits required to address registers.
Two source registers and one destination register
→ therefore ,dual ported register file
supporting two reads and one write

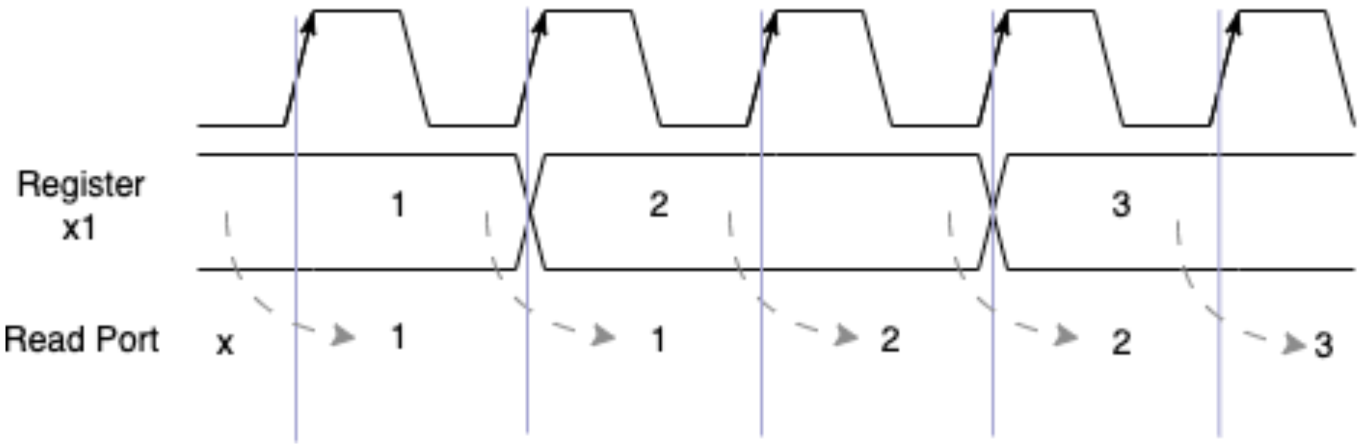


Register bank is a collection of registers along with muxes and a demux.
In a single clock cycle, it can read two registers and write to a single register.

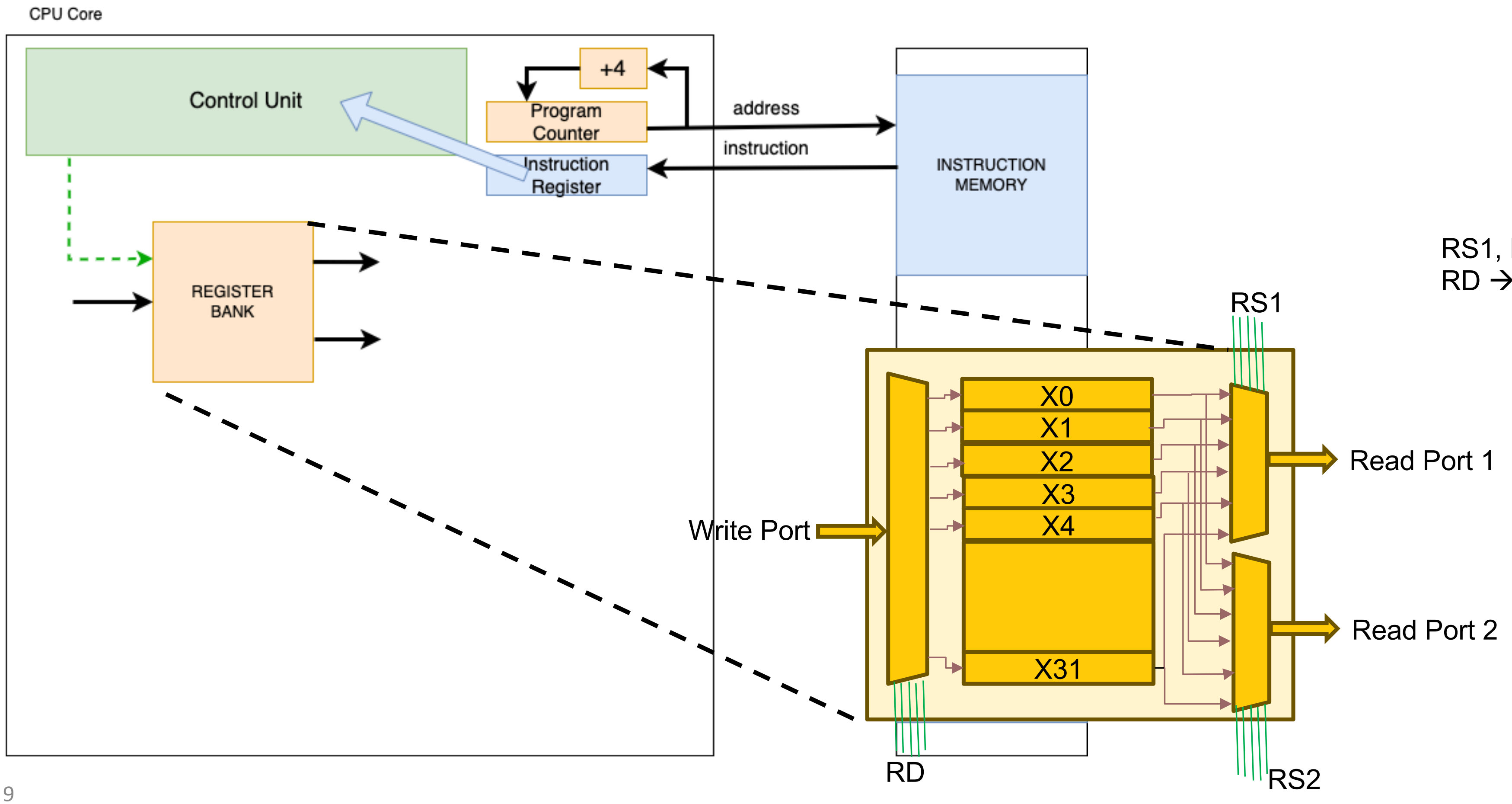
Example of Dual Ported Register



Timing diagram of how data gets read out



The Register Bank in the CPU

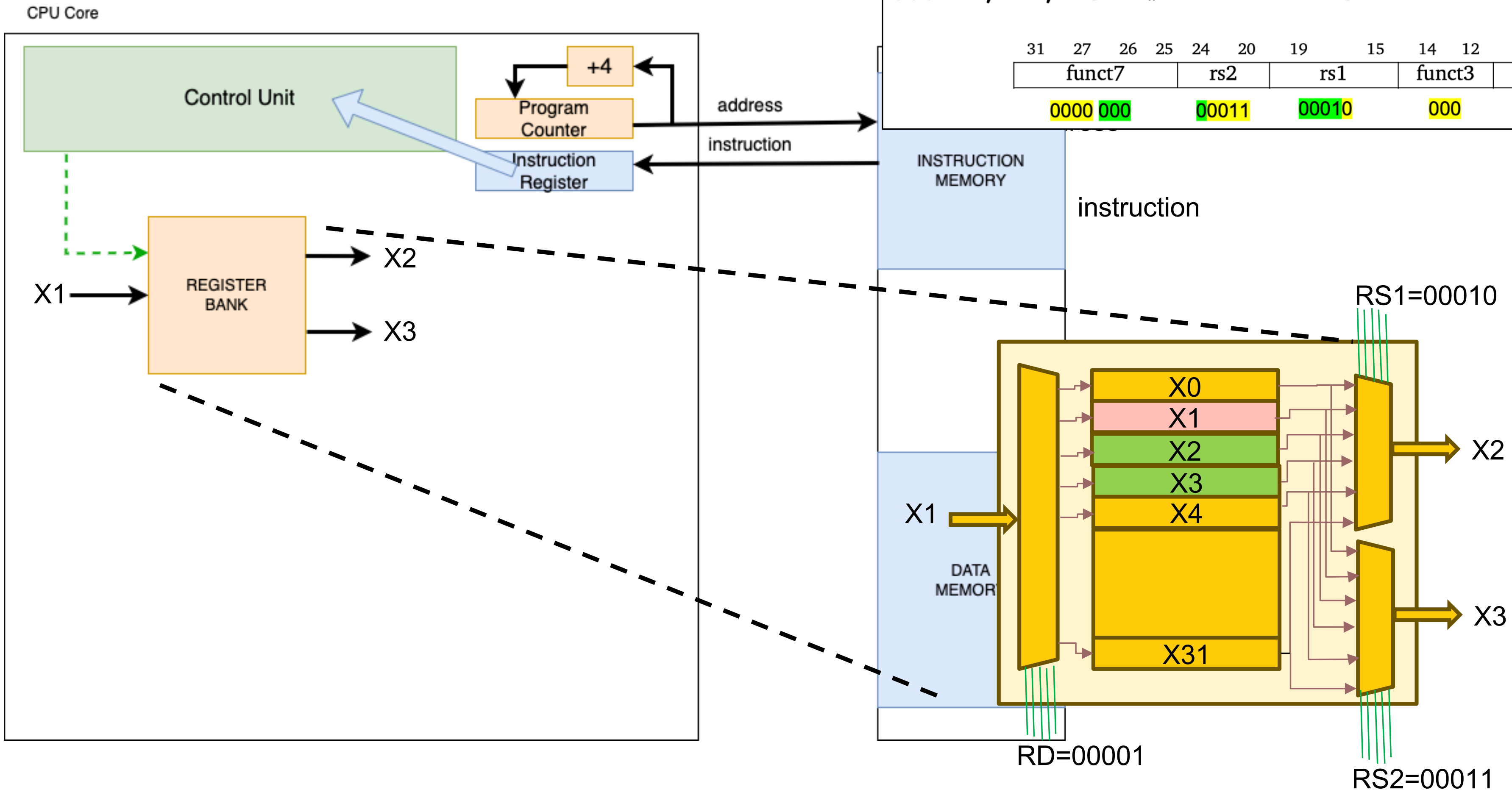


RS1, RS2 → source registers
RD → destination register

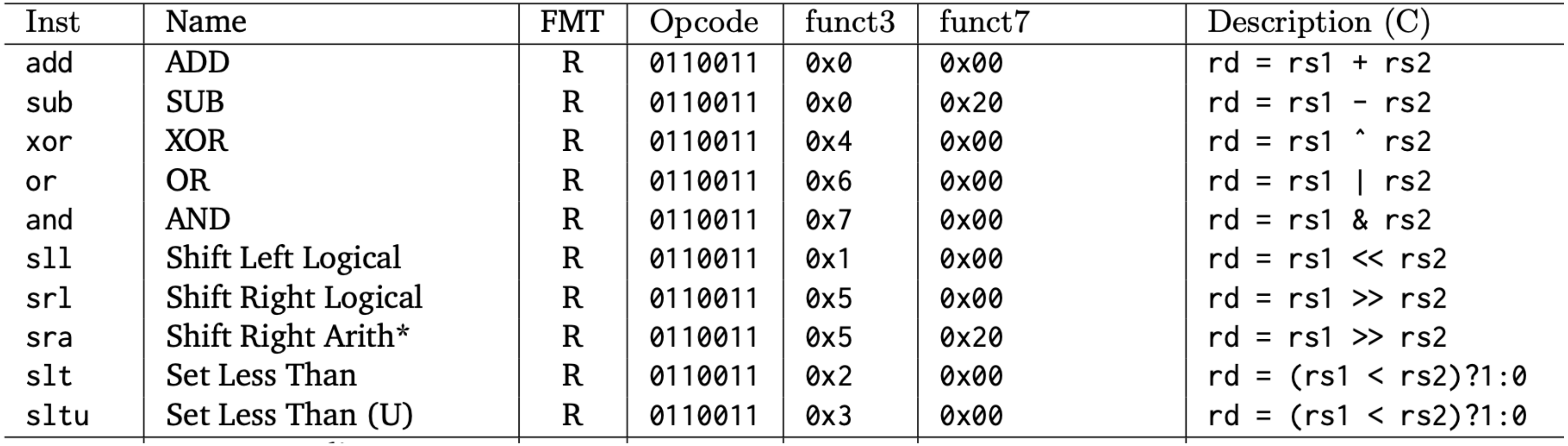
The Register Bank : Example

```
add x1, x2, x3    # x1 = x2 + x3
```

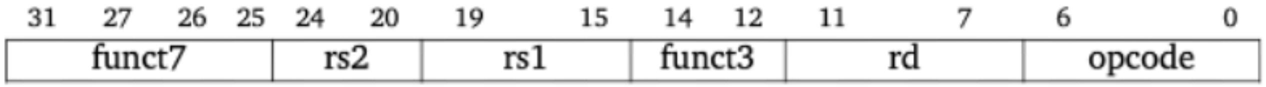
31 27 26 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
0000 000	00011	00010	000	00001	0110011



R-Type Instructions



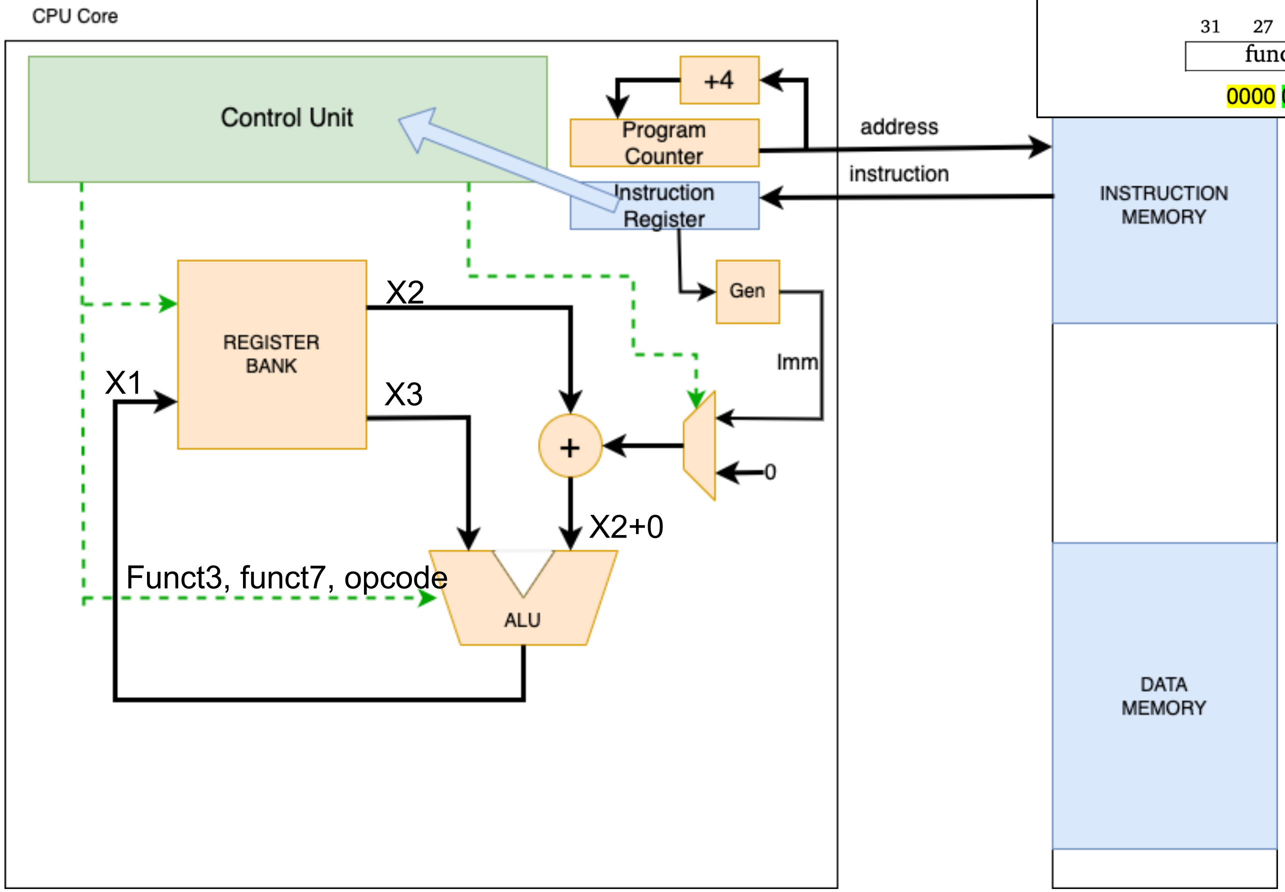
Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ˆ rs2
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0



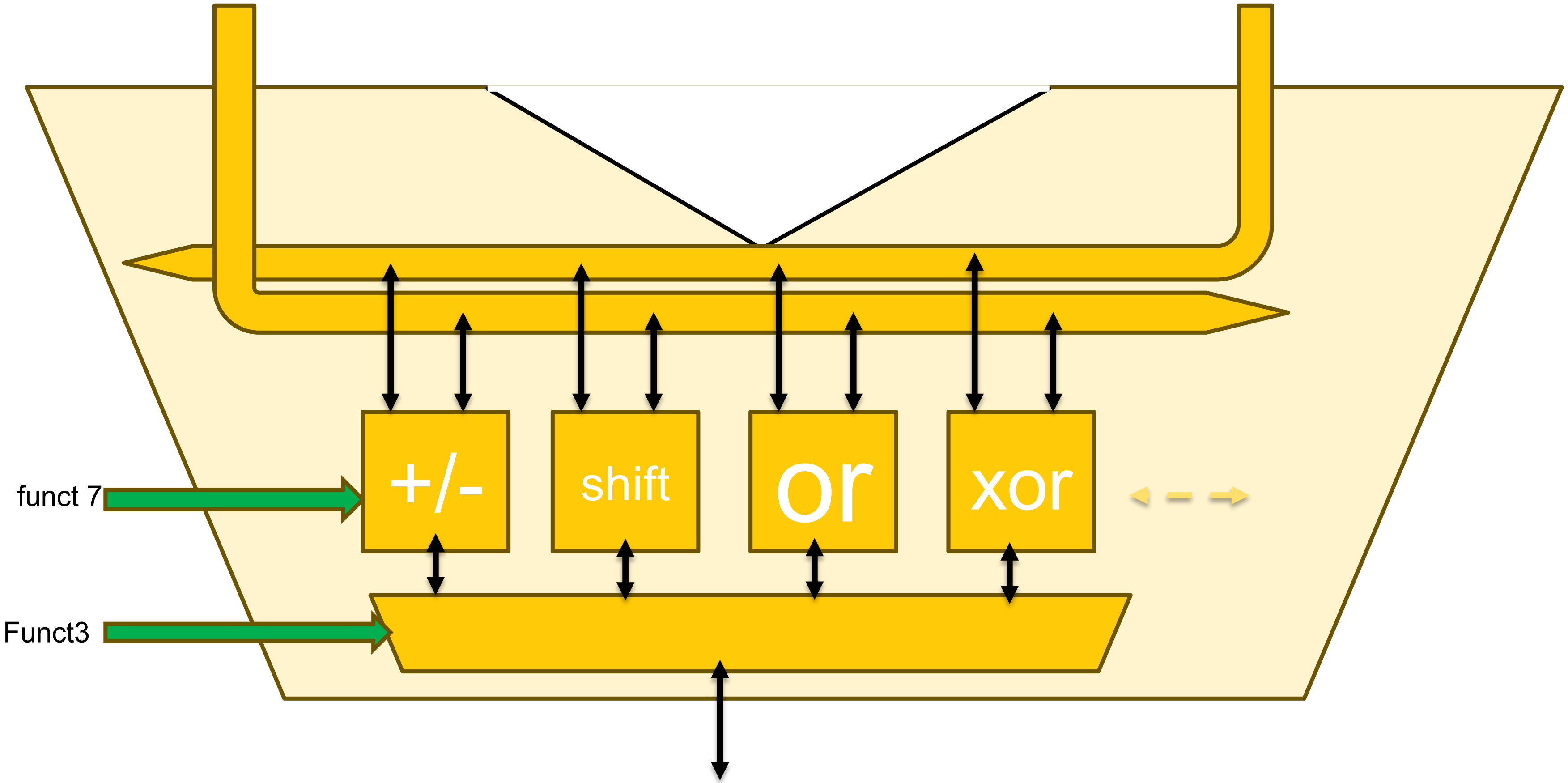
Arithmetic Operations (R type instructions)

```
add x1, x2, x3    # x1 = x2 + x3
```

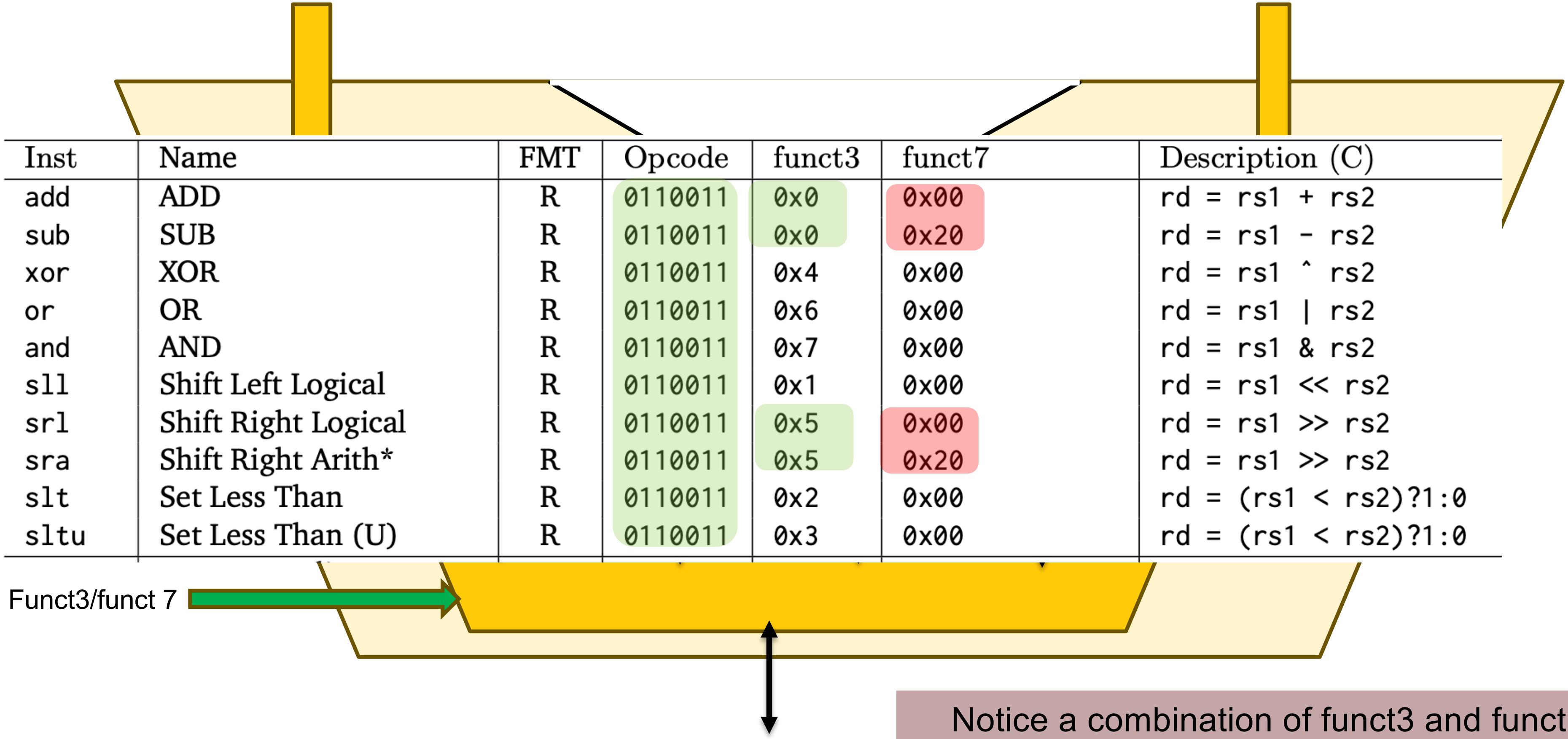
31 27 26 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
0000 000	00011	00010	000	00001	0110011



Inside the ALU



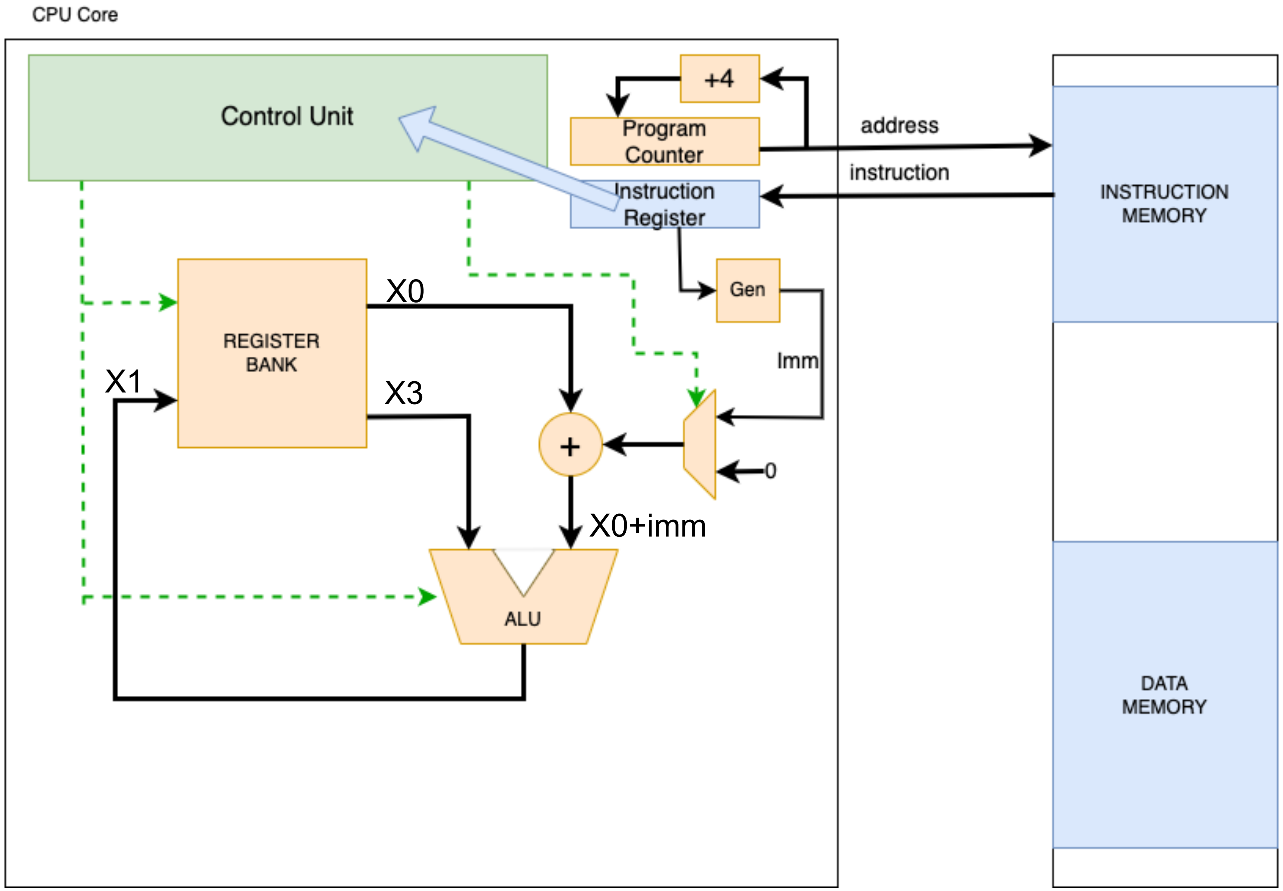
Inside the ALU



Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ˆ rs2
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0

Notice a combination of funct3 and funct7 uniquely distinguishes between the instructions

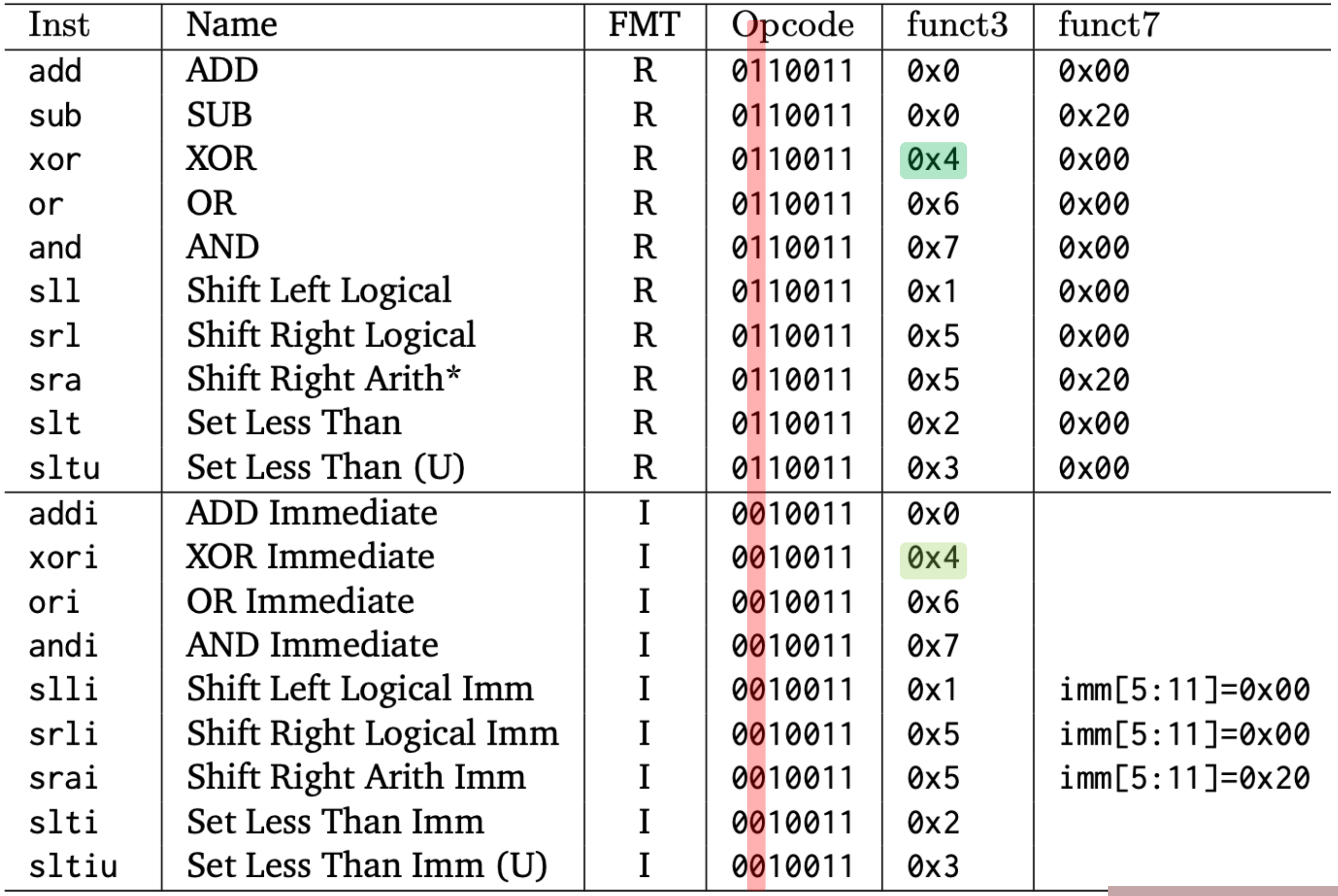
Arithmetic Operations (I type instructions)



```
addi x1, x3, imm
```

What is the use of Gen?

Comparing the encoding of R and I type instructions

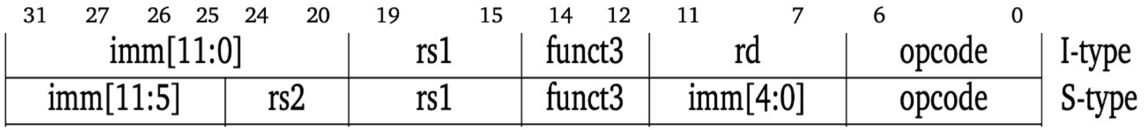


Inst	Name	FMT	Opcode	funct3	funct7
add	ADD	R	0110011	0x0	0x00
sub	SUB	R	0110011	0x0	0x20
xor	XOR	R	0110011	0x4	0x00
or	OR	R	0110011	0x6	0x00
and	AND	R	0110011	0x7	0x00
sll	Shift Left Logical	R	0110011	0x1	0x00
srl	Shift Right Logical	R	0110011	0x5	0x00
sra	Shift Right Arith*	R	0110011	0x5	0x20
slt	Set Less Than	R	0110011	0x2	0x00
sltu	Set Less Than (U)	R	0110011	0x3	0x00
addi	ADD Immediate	I	0010011	0x0	
xori	XOR Immediate	I	0010011	0x4	
ori	OR Immediate	I	0010011	0x6	
andi	AND Immediate	I	0010011	0x7	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20
slti	Set Less Than Imm	I	0010011	0x2	
sltiu	Set Less Than Imm (U)	I	0010011	0x3	

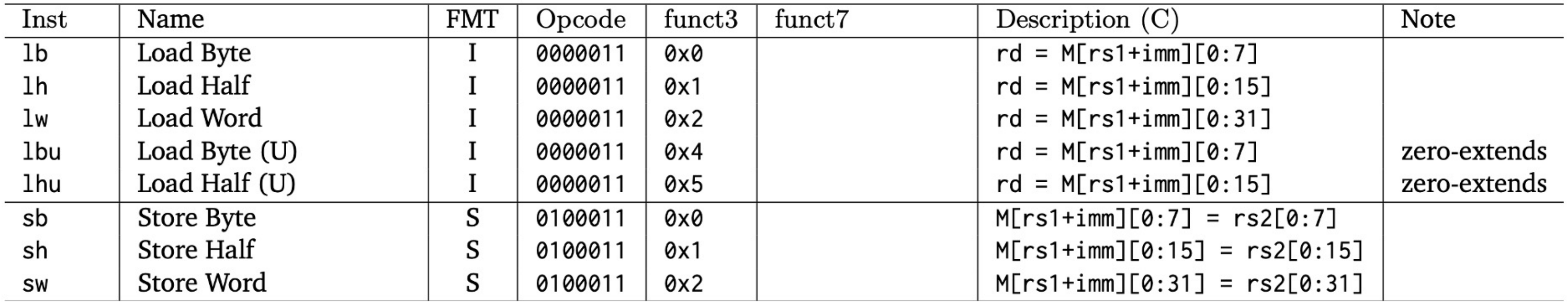
Notice the similarities and differences

Load and Store Instructions

- Are of two types (I and S)

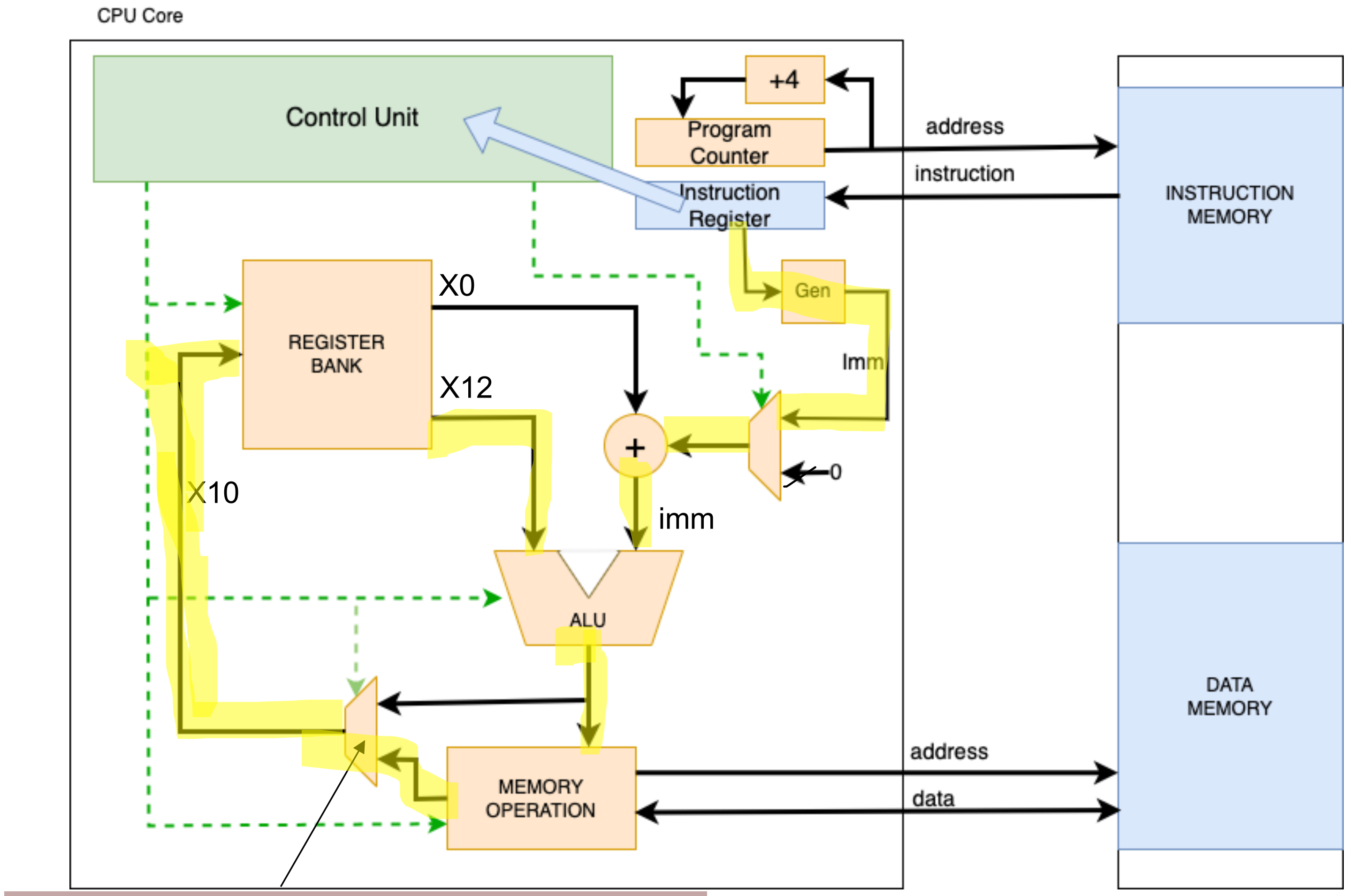


31 27 26 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type



Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	

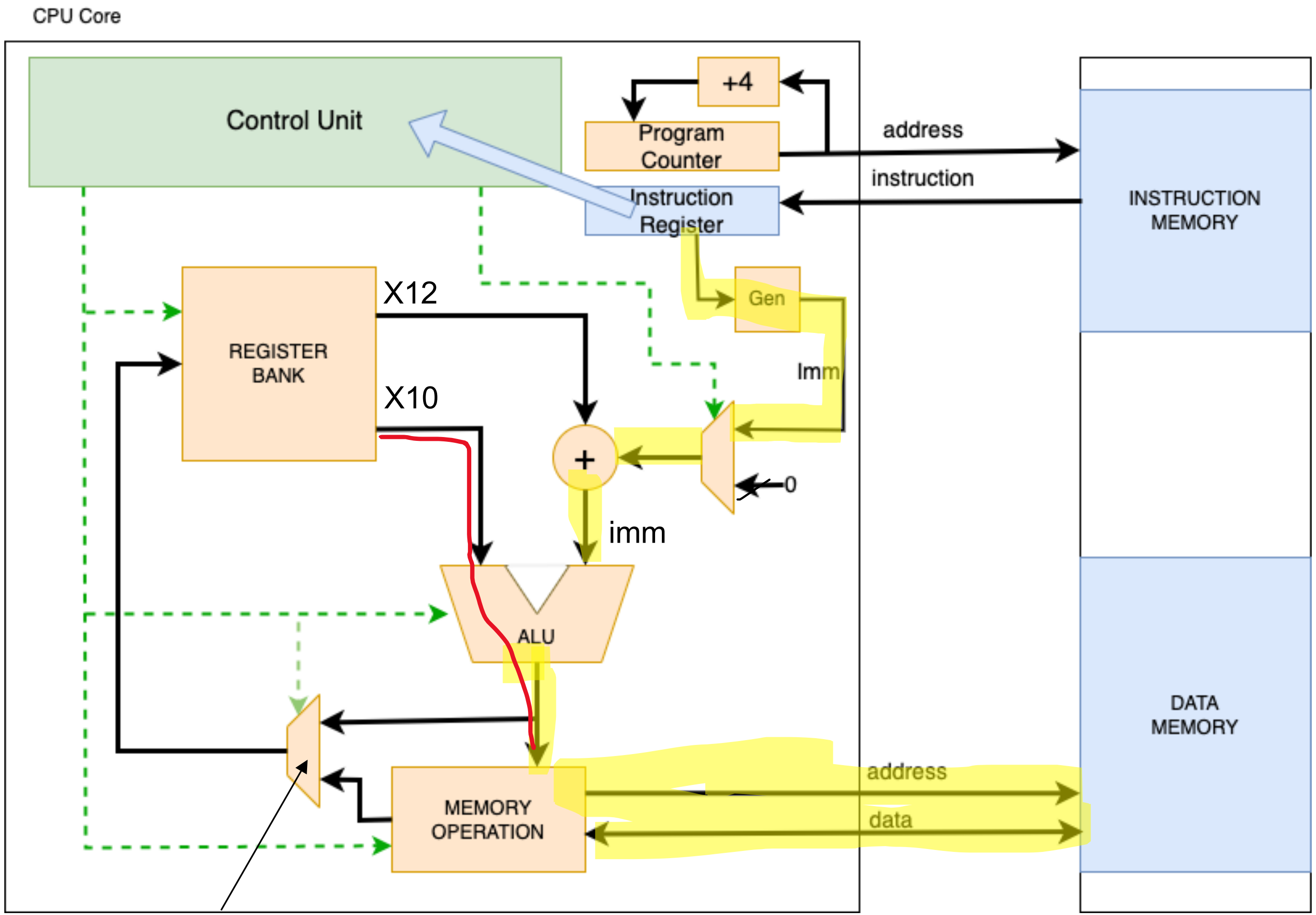
Load (I type instructions)



```
lw x10, imm(x12)
```

distinguish between ALU and memory operations?

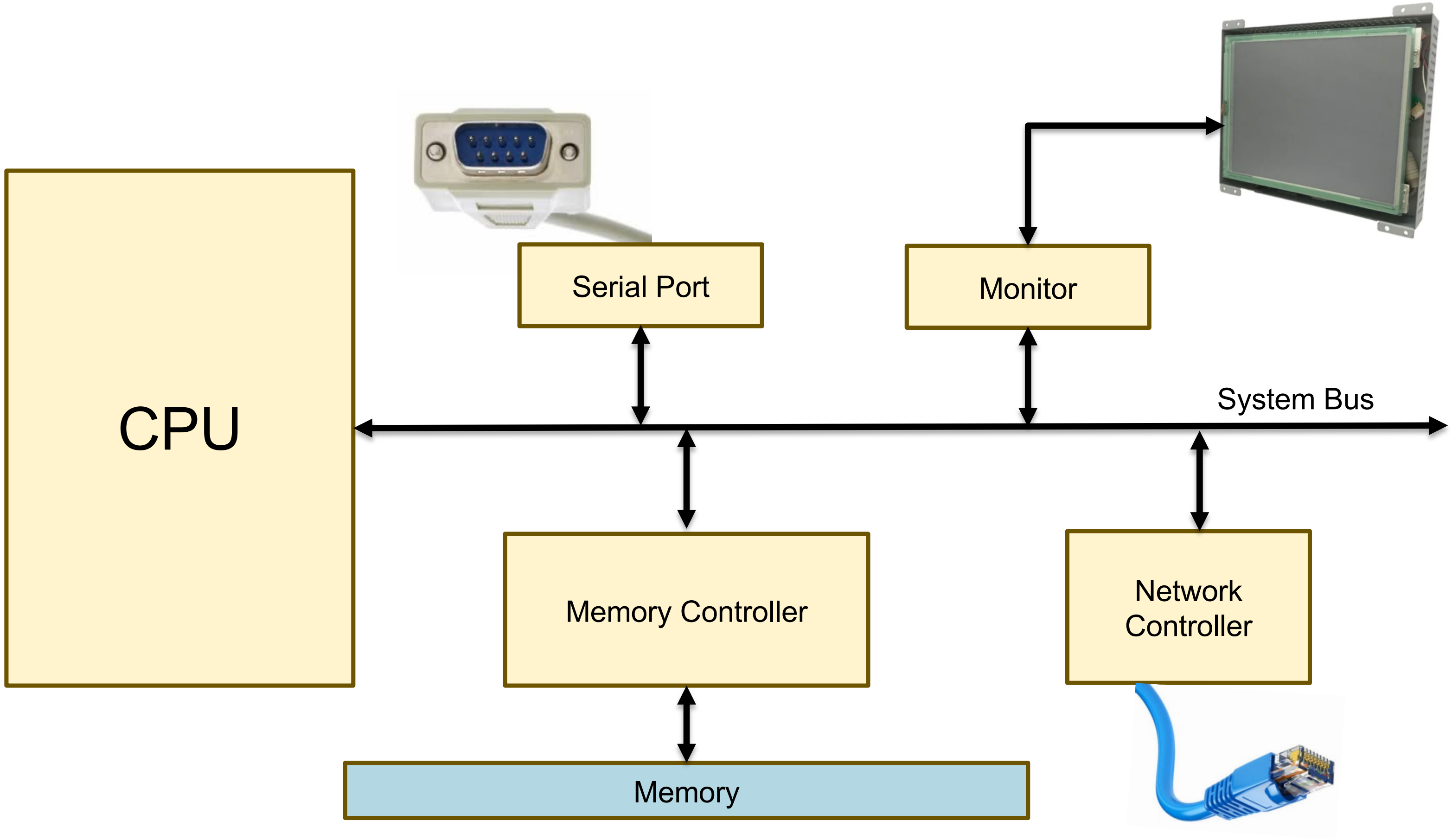
Store (S type instructions)



```
sw x10, imm(x12)
```

distinguish between ALU and memory operations?

Memory Mapped Peripherals

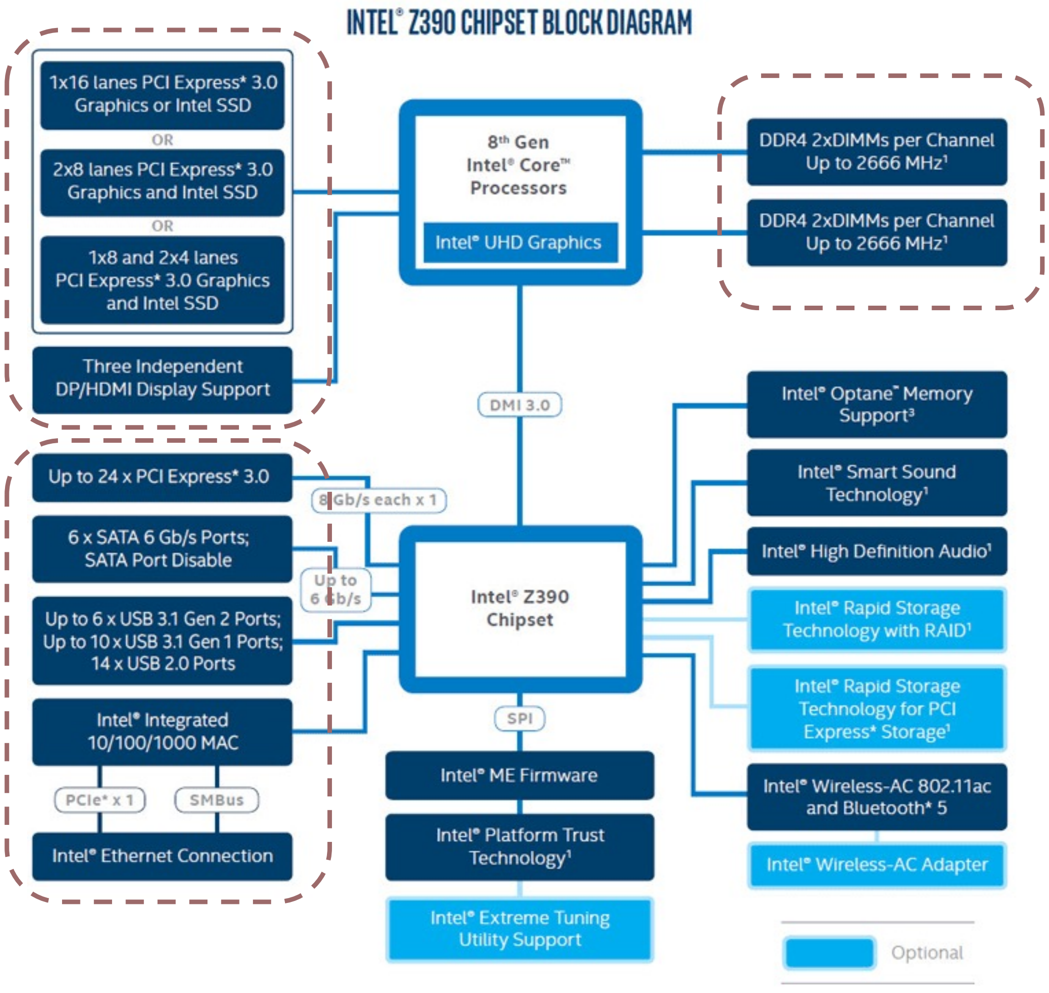


Not just memory, peripherals are also connected to the CPU

Example of a Intel motherboard block diagram

High Speed Peripherals

Low Speed Peripherals



INTEL® Z390 CHIPSET BLOCK DIAGRAM

1x16 lanes PCI Express* 3.0 Graphics or Intel SSD

OR

2x8 lanes PCI Express* 3.0 Graphics and Intel SSD

OR

1x8 and 2x4 lanes PCI Express* 3.0 Graphics and Intel SSD

Three Independent DP/HDMI Display Support

8th Gen Intel® Core™ Processors

Intel® UHD Graphics

DDR4 2xDIMMs per Channel Up to 2666 MHz[1]

DDR4 2xDIMMs per Channel Up to 2666 MHz[1]

DMI 3.0

Up to 24 x PCI Express* 3.0

8 Gb/s each x 1

6 x SATA 6 Gb/s Ports; SATA Port Disable

Up to 6 Gb/s

Up to 6 x USB 3.1 Gen 2 Ports; Up to 10 x USB 3.1 Gen 1 Ports; 14 x USB 2.0 Ports

Intel® Integrated 10/100/1000 MAC

PCIe* x 1

SMBus

Intel® Ethernet Connection

Intel® Z390 Chipset

SPI

Intel® ME Firmware

Intel® Platform Trust Technology[1]

Intel® Extreme Tuning Utility Support

Intel® Optane™ Memory Support[3]

Intel® Smart Sound Technology[1]

Intel® High Definition Audio[1]

Intel® Rapid Storage Technology with RAID[1]

Intel® Rapid Storage Technology for PCI Express* Storage[1]

Intel® Wireless-AC 802.11ac and Bluetooth* 5

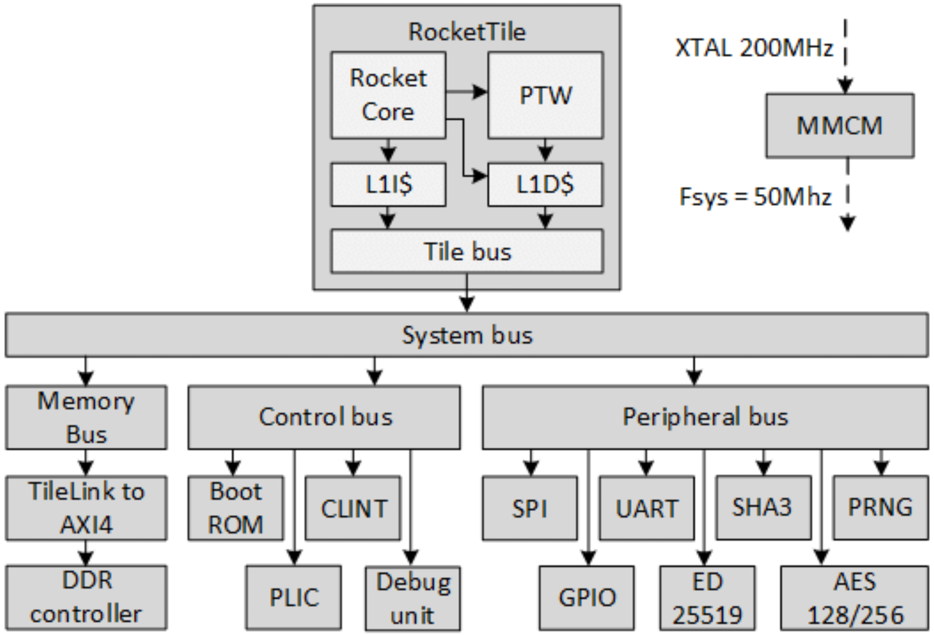
Intel® Wireless-AC Adapter

Optional

Memory

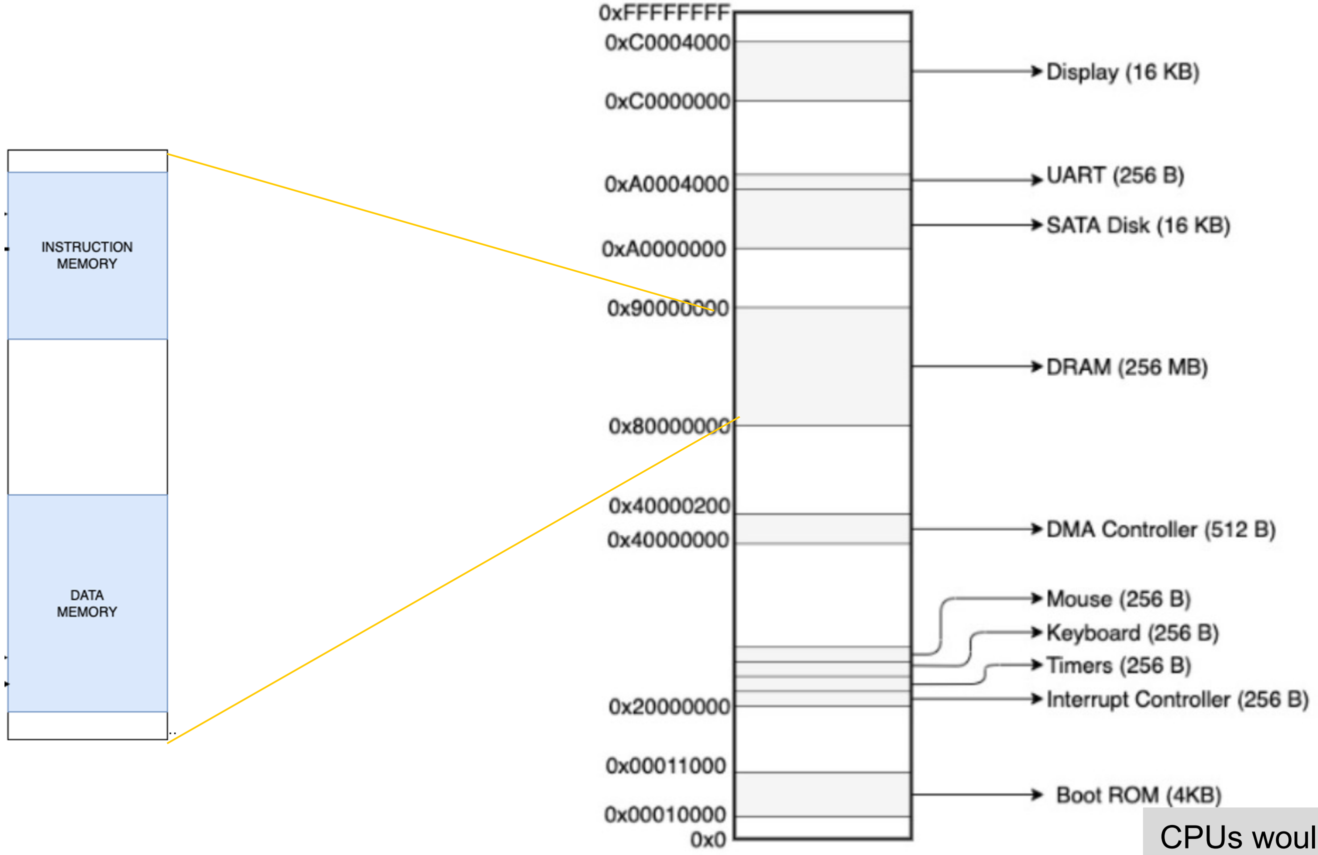
Notice that besides memory, lot of other peripherals connected to the CPU (Intel core)

Example of a RISC V System on Chip



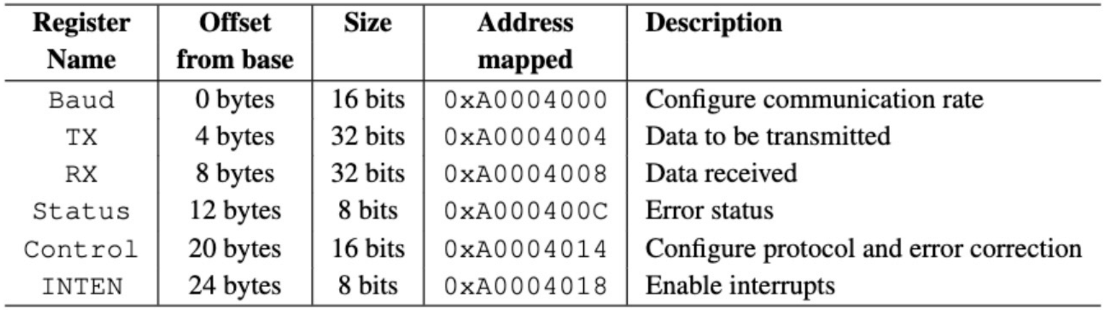
Addressing Memory Mapped Peripherals

Entire Memory Range of CPU (for example, 2^{32})



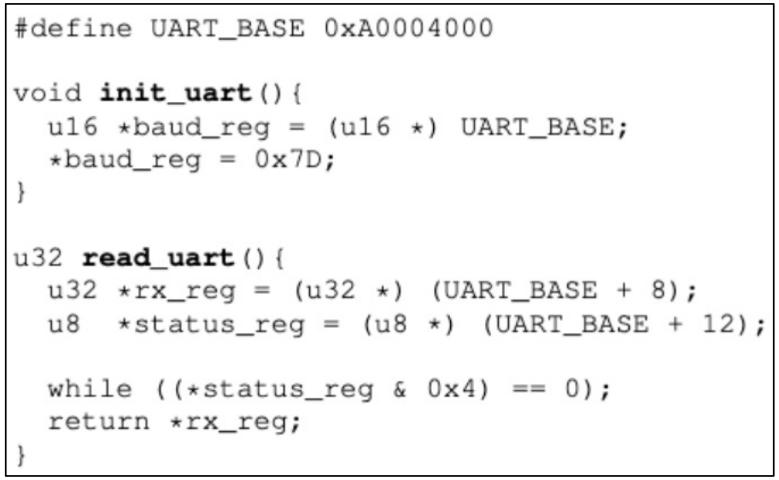
CPUs would have a unique address map to identify memory and peripherals.

Serial Port (UART) Example

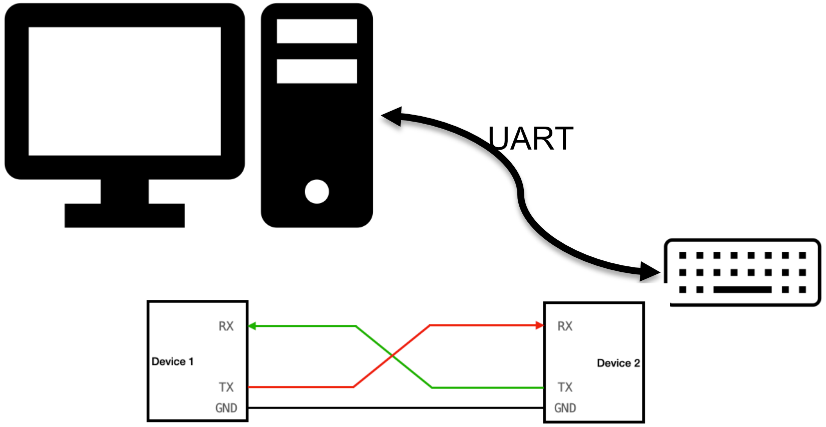


Register Name	Offset from base	Size	Address mapped	Description
Baud	0 bytes	16 bits	0xA0004000	Configure communication rate
TX	4 bytes	32 bits	0xA0004004	Data to be transmitted
RX	8 bytes	32 bits	0xA0004008	Data received
Status	12 bytes	8 bits	0xA000400C	Error status
Control	20 bytes	16 bits	0xA0004014	Configure protocol and error correction
INTEN	24 bytes	8 bits	0xA0004018	Enable interrupts

Functions to read and initialize UART

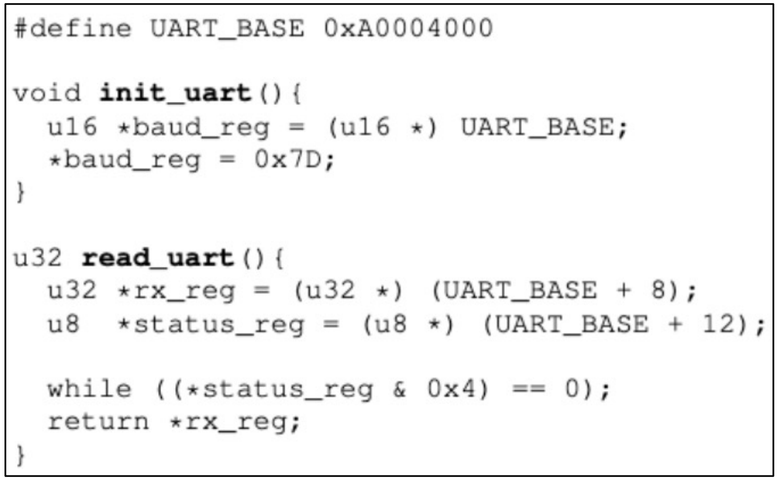


```
#define UART_BASE 0xA0004000

void init_uart(){
  u16 *baud_reg = (u16 *) UART_BASE;
  *baud_reg = 0x7D;
}

u32 read_uart(){
  u32 *rx_reg = (u32 *) (UART_BASE + 8);
  u8  *status_reg = (u8 *) (UART_BASE + 12);

  while ((*status_reg & 0x4) == 0);
  return *rx_reg;
}
```



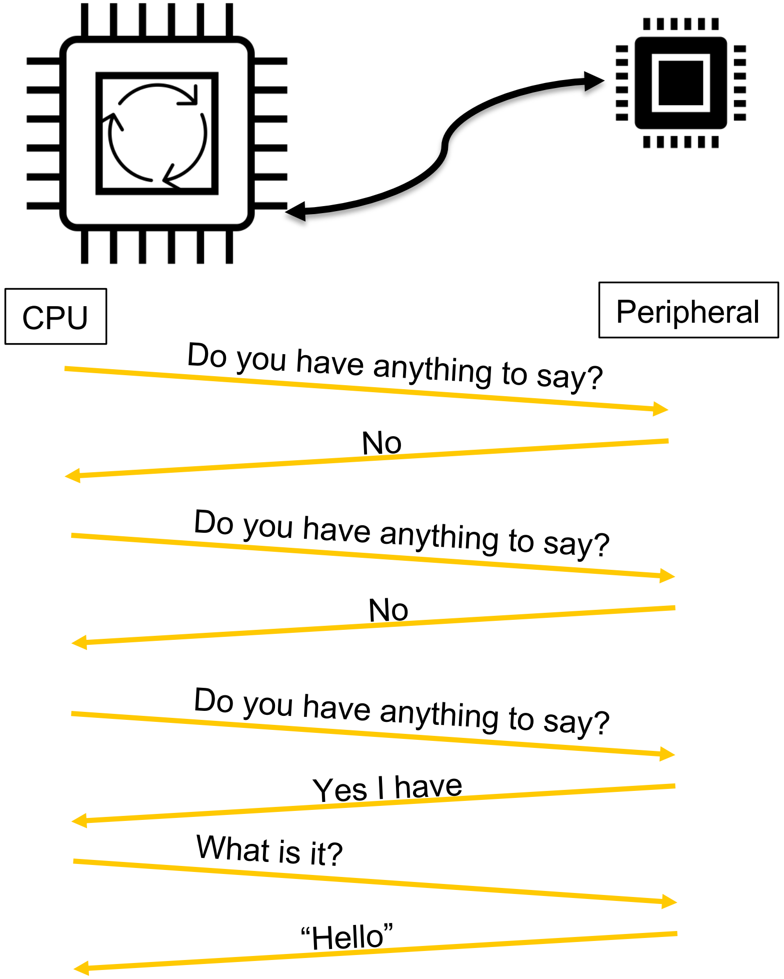
Peripherals have a set of memory mapped locations which can be read/written to by the CPU to communicate with the peripheral.

Program to read from the UART using Polling



```
#define UART_BASE 0xA0004000

void init_uart(){
  u16 *baud_reg = (u16 *) UART_BASE;
  *baud_reg = 0x7D;
}

u32 read_uart(){
  u32 *rx_reg = (u32 *)  (UART_BASE + 8);
  u8  *status_reg = (u8 *)  (UART_BASE + 12);

  while ((*status_reg & 0x4) == 0);
  return *rx_reg;
}
```

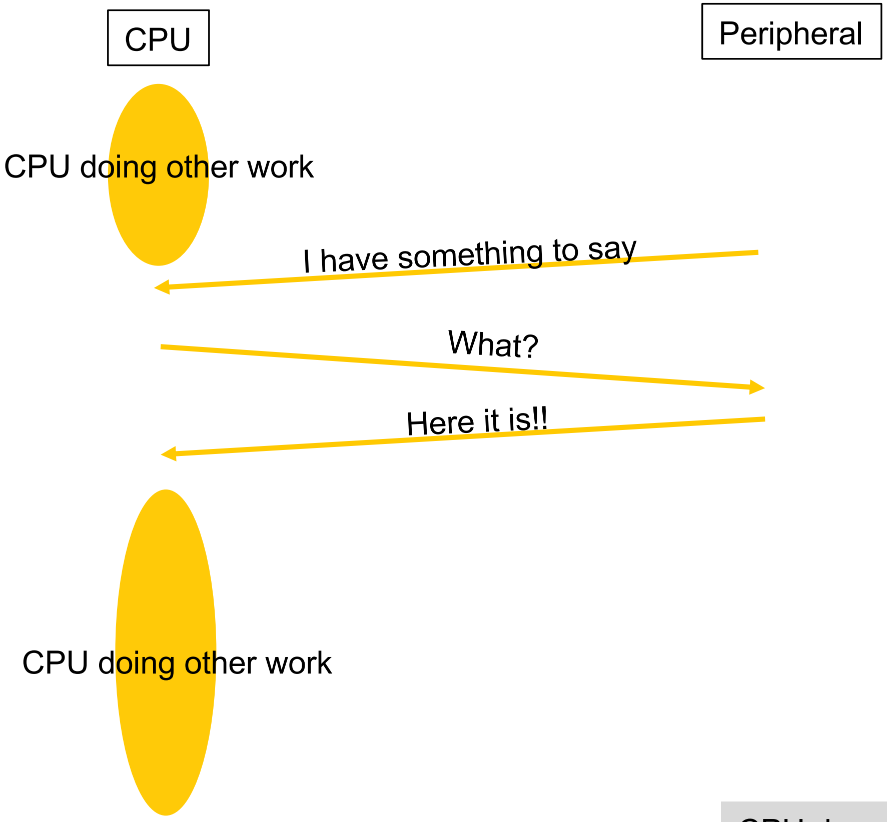
```
void main(){
   u32 uartrx;

   while(1){
      uartrx = read_uart();
      if (uartrx != 0) break;
   }

   // process uartrx here
}
```


CPU
Peripheral
Do you have anything to say?
No
Do you have anything to say?
No
Do you have anything to say?
Yes I have
What is it?
"Hello"

Problem!! If nothing is received, CPU wastes time in the while loop doing nothing useful

Interrupts

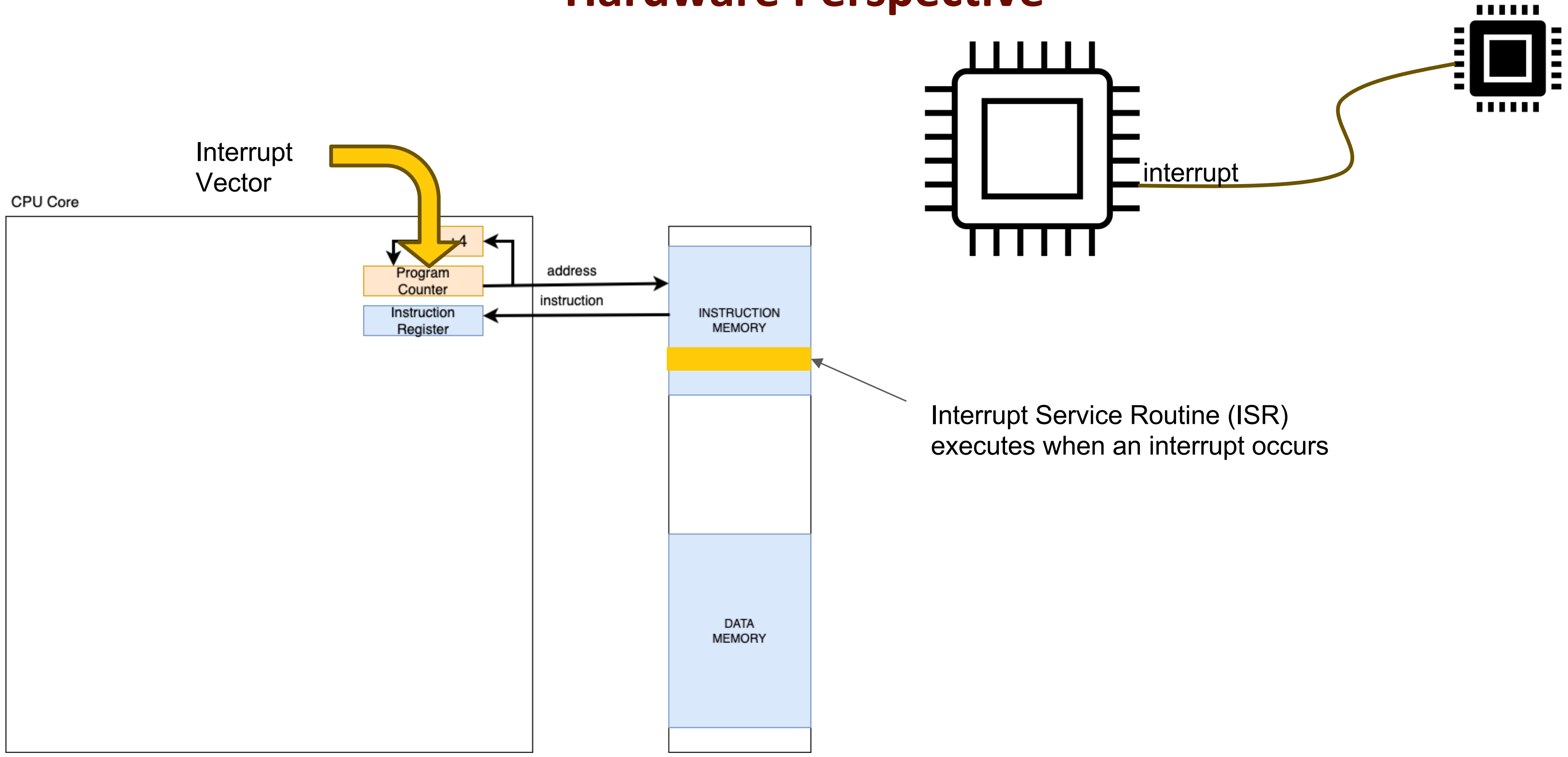
Peripheral sends an interrupt whenever it wants to communicate with the CPU



CPU does other work and only communicates with the peripheral when there is an interrupt

How do interrupts work?

Hardware Perspective

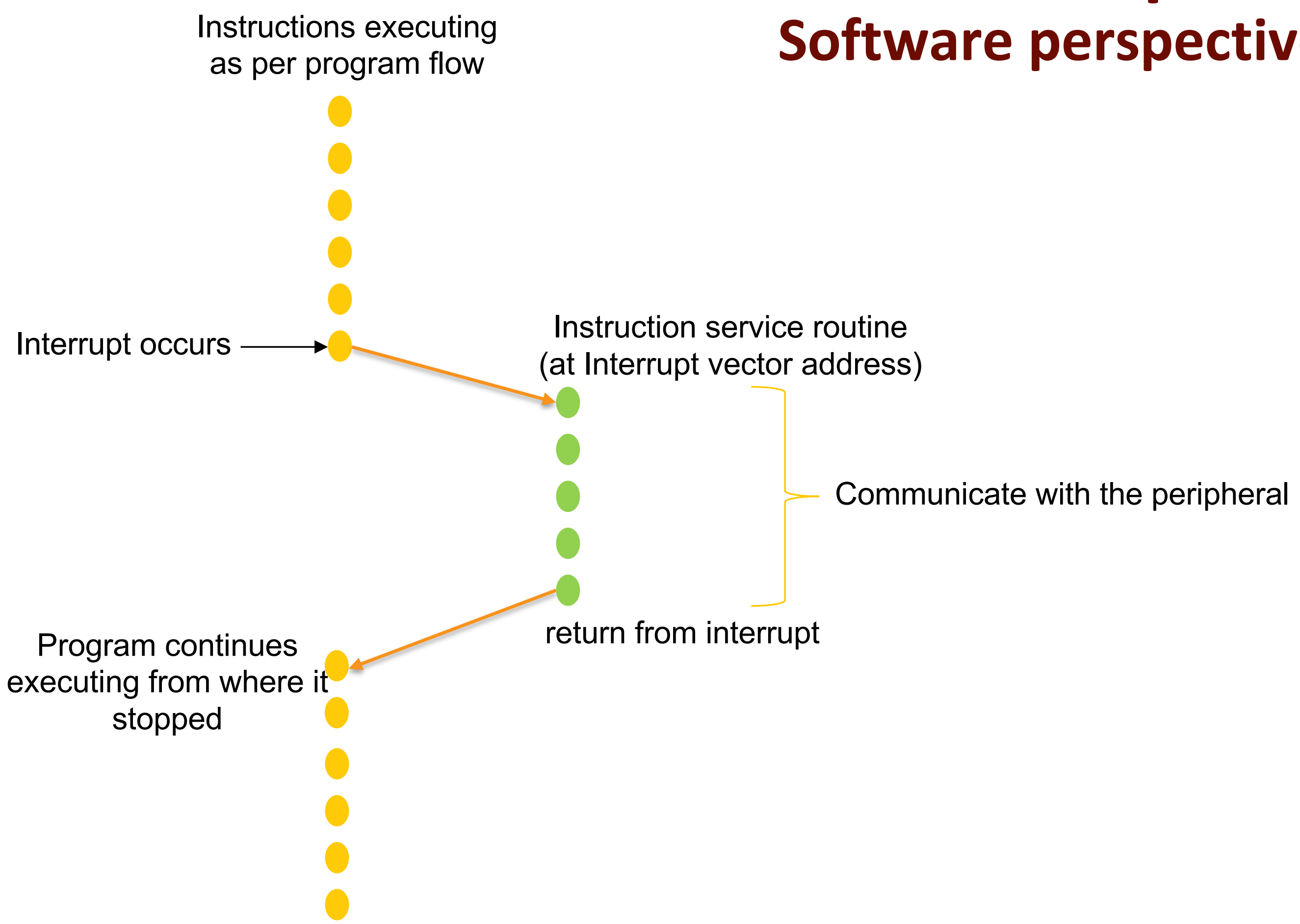


Interrupt Service Routine (ISR) executes when an interrupt occurs

Peripheral raises an interrupt → PC changed to Interrupt Vector → ISR executes

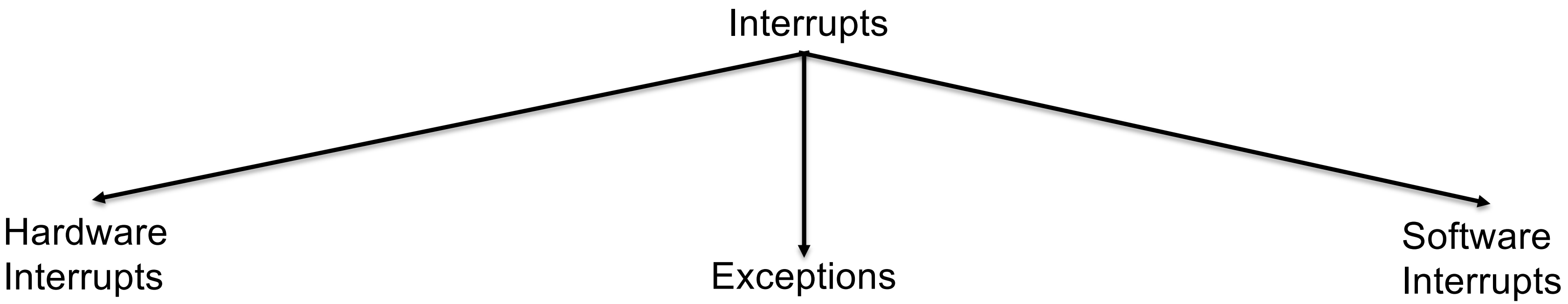
How do interrupts work?

Software perspective


Instructions executing as per program flow
Interrupt occurs
Instruction service routine (at Interrupt vector address)
Communicate with the peripheral
return from interrupt
Program continues executing from where it stopped

Peripheral raises an interrupt → PC changed to Interrupt Vector → ISR executes → when ISR completes, previous program continues to execute

Types of Interrupts



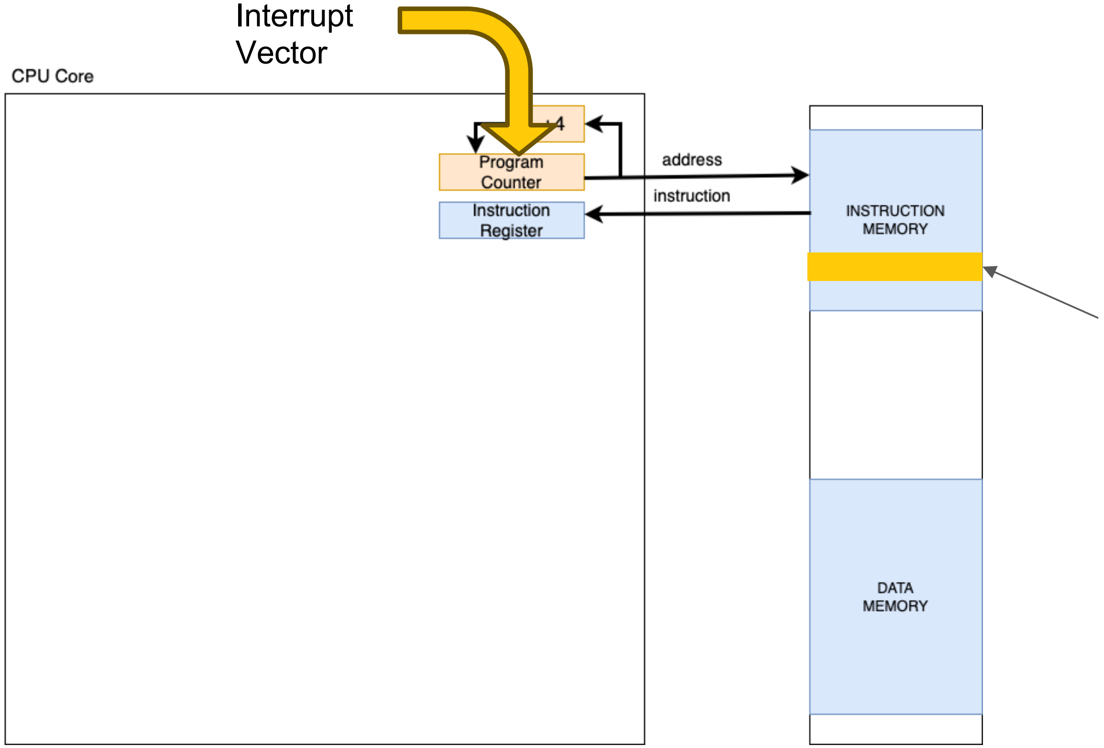
Hardware Interrupts (aka Interrupt Requests IRQ, External Interrupts) … is from external peripherals

Software Interrupt is triggered with an explicit instruction (like `int 0x80`), that executes in the program.

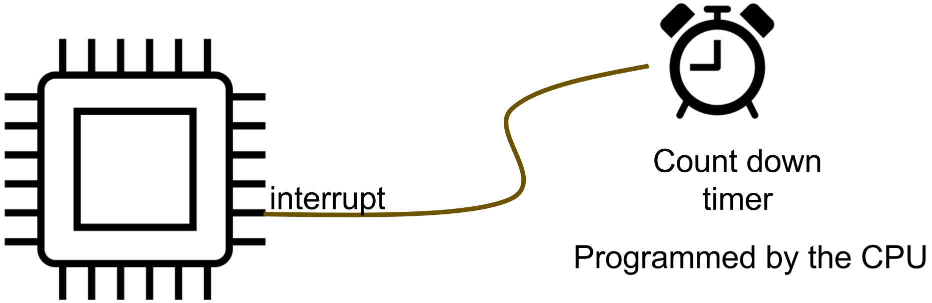
Exceptions, triggered by the CPU itself when there is a major problem: eg. Divide by zero, illegal instruction, hardware malfunction, memory not present, etc.

Use of Hardware Interrupts (Example)

Timers



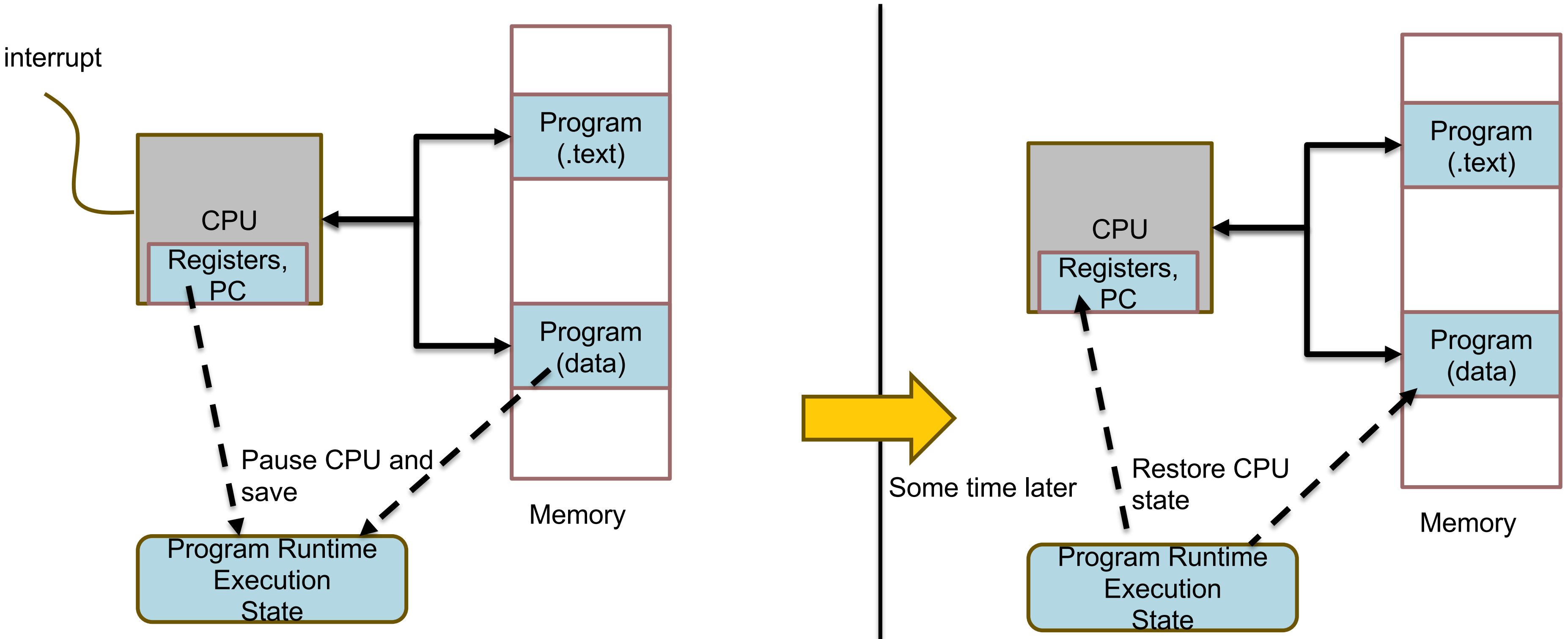
Timer Routine (ISR)
executes when the timer interrupt occurs



Use Timer Interrupts to pause a program execution and continue it later.

Programmed by the CPU, the timer sends an interrupt at the end of the count (ie count becomes 0)

State of an Executing Program



interrupt

CPU

Registers, PC

Program (.text)

Program (data)

Memory

Pause CPU and save

Program Runtime Execution State

Some time later

CPU

Registers, PC

Program (.text)

Program (data)

Memory

Restore CPU state

Program Runtime Execution State

The “Program Runtime Execution State” holds the registers, PC and data memory at time instant t, when the interrupt occurs

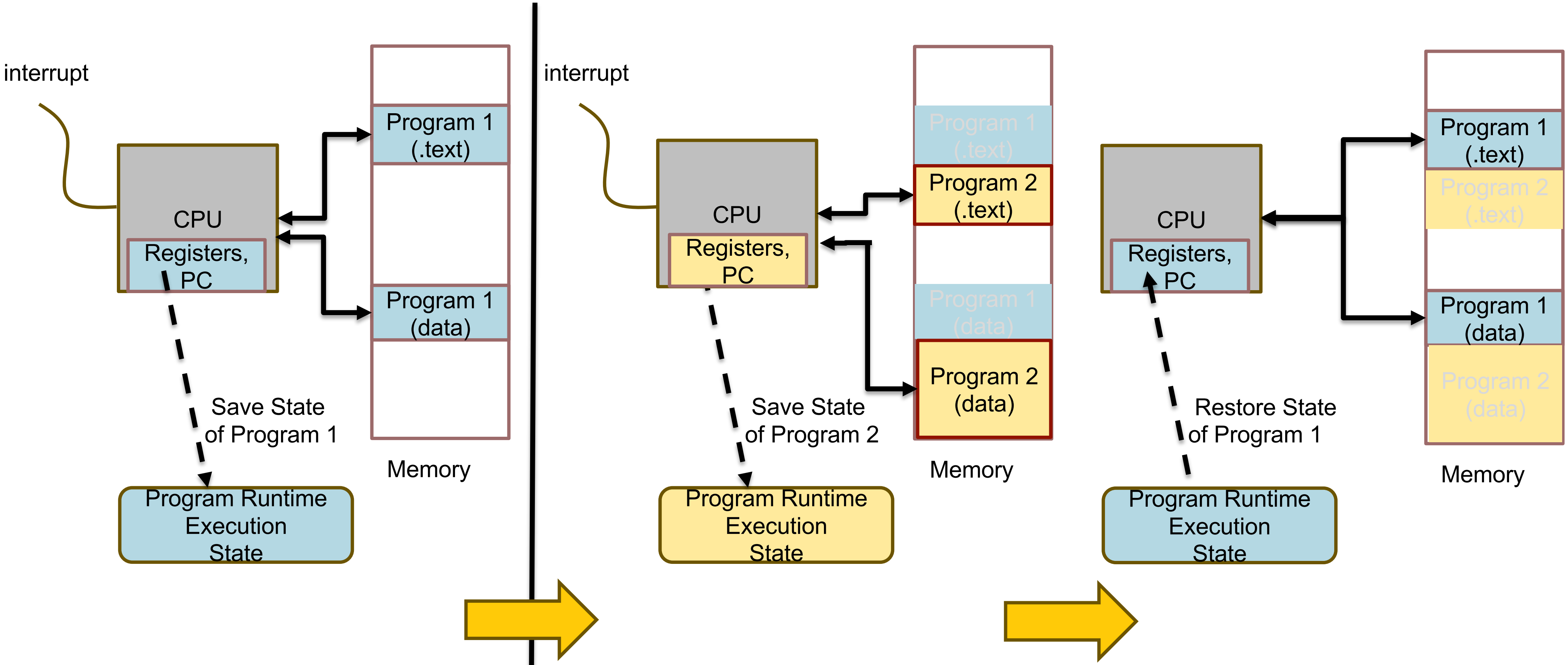
At a time later, restore the “Program Runtime Execution State” to continue the execution from time instant t.

When is saving / restoring the state useful?

- Example. Close the top cover (display) of laptop

- Another example: Multi-tasking, execute multiple programs simultaneously.

Multi-tasking program execution

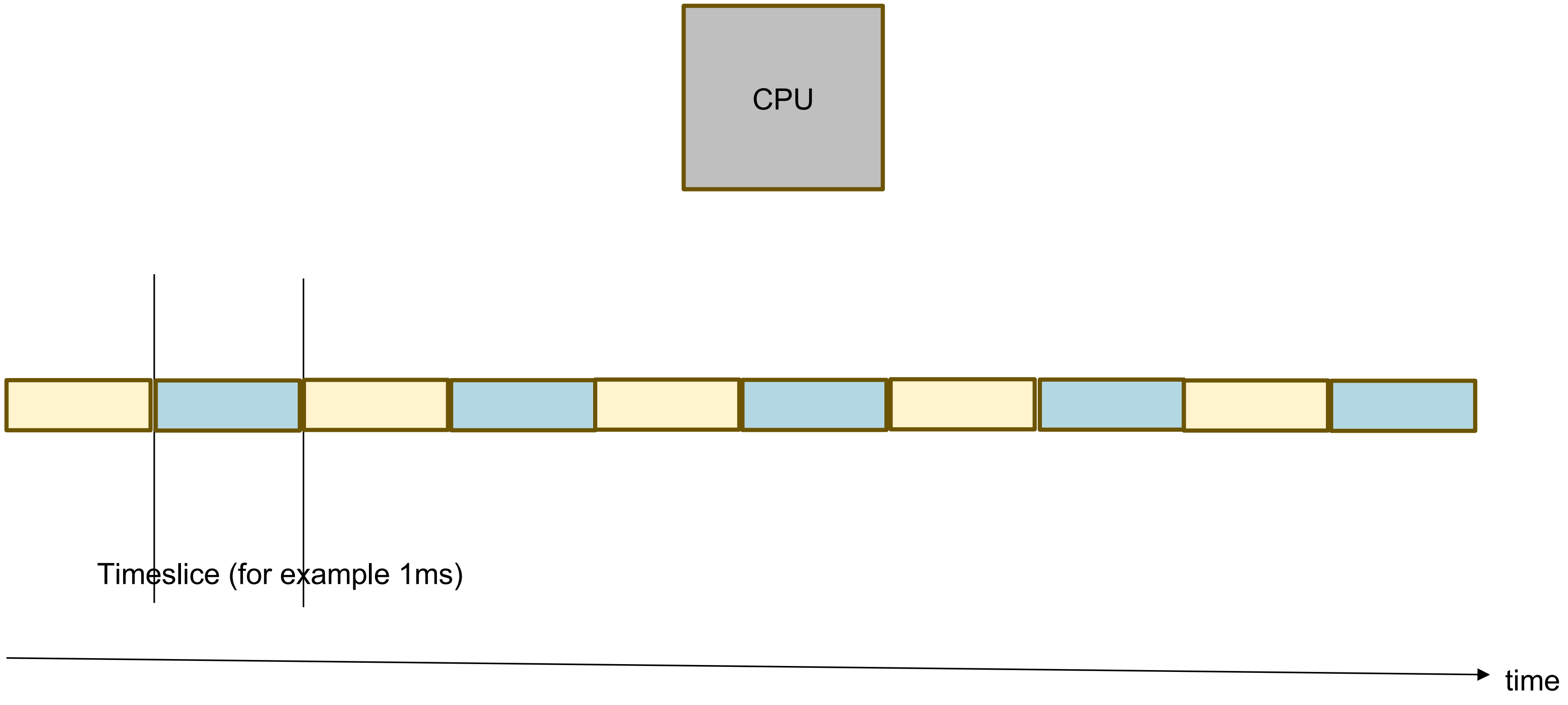


Program 1 is executing in CPU. On interrupt, state of Program 1 is stored

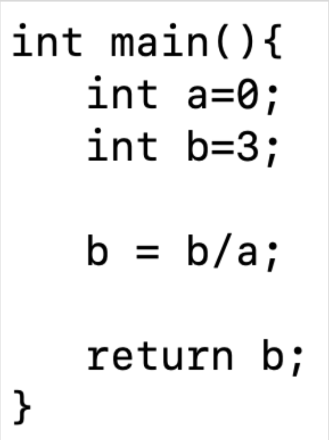
Program 2 is executing in CPU. On interrupt, state of Program 2 is stored

… and state of Program 1 is restored. Program 1 continues executing from where it stopped.

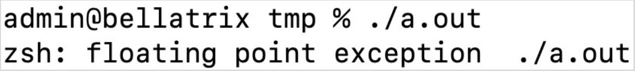
Multitasking time view



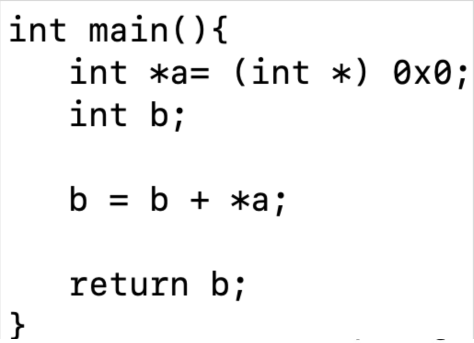
Use of Exceptions (Examples)



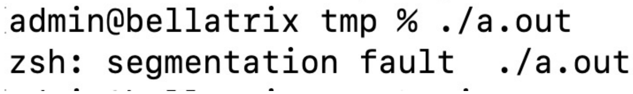
```
int main(){
   int a=0;
   int b=3;

   b = b/a;

   return b;
}
```



```
admin@bellatrix tmp % ./a.out
zsh: floating point exception  ./a.out
```



```
int main(){
   int *a= (int *) 0x0;
   int b;

   b = b + *a;

   return b;
}
```



```
admin@bellatrix tmp % ./a.out
zsh: segmentation fault  ./a.out
```

When there is a problem that occurs when the program Executes.