A First Look at Microprocessors

using the

The General Prototype Computer (GPC) model

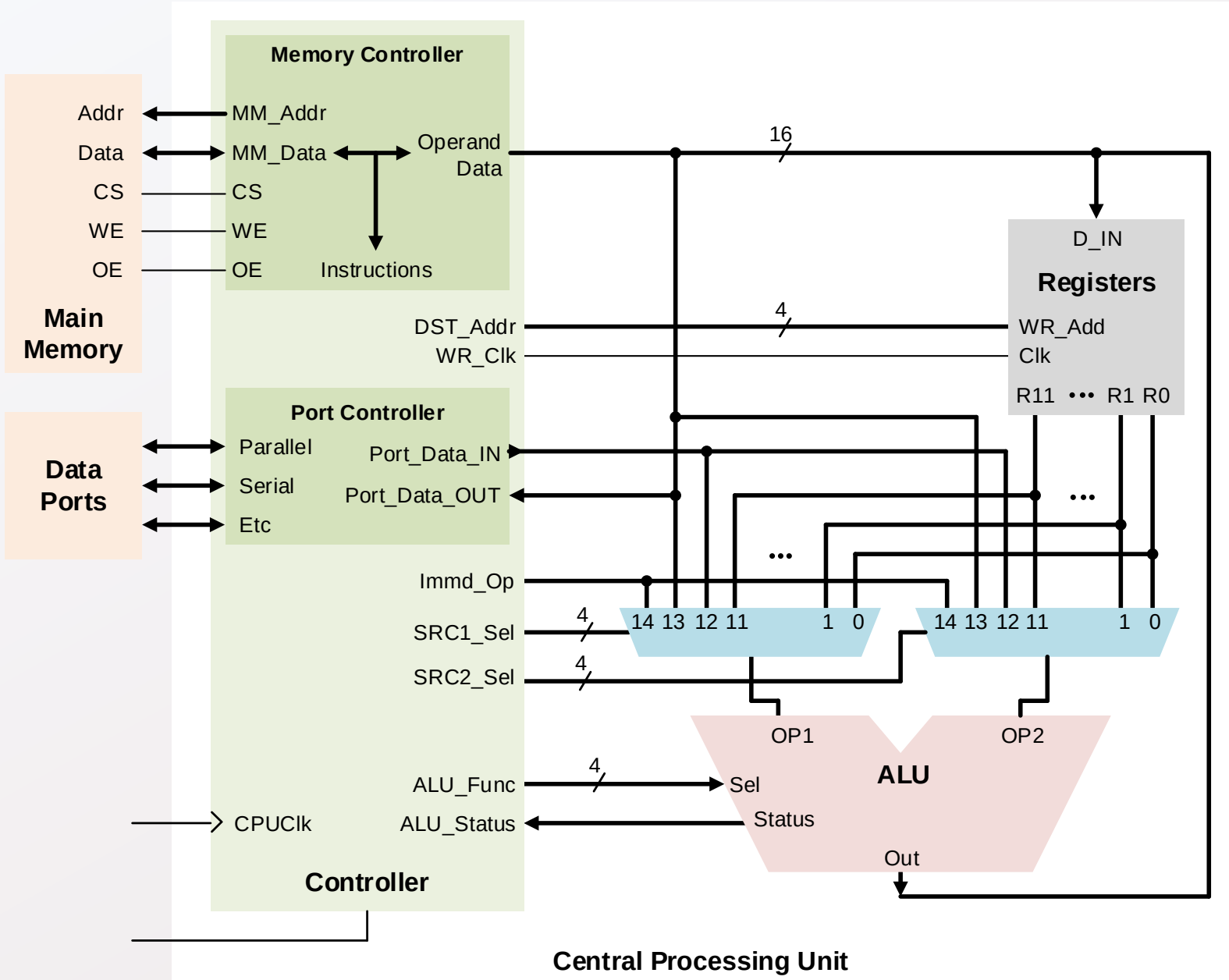
Part 2

Can you identify an opcode to:

Decrement the contents of R1, and store the result in R5?

Invert the contents of R2, and store the result back in R2? (Will this work?)

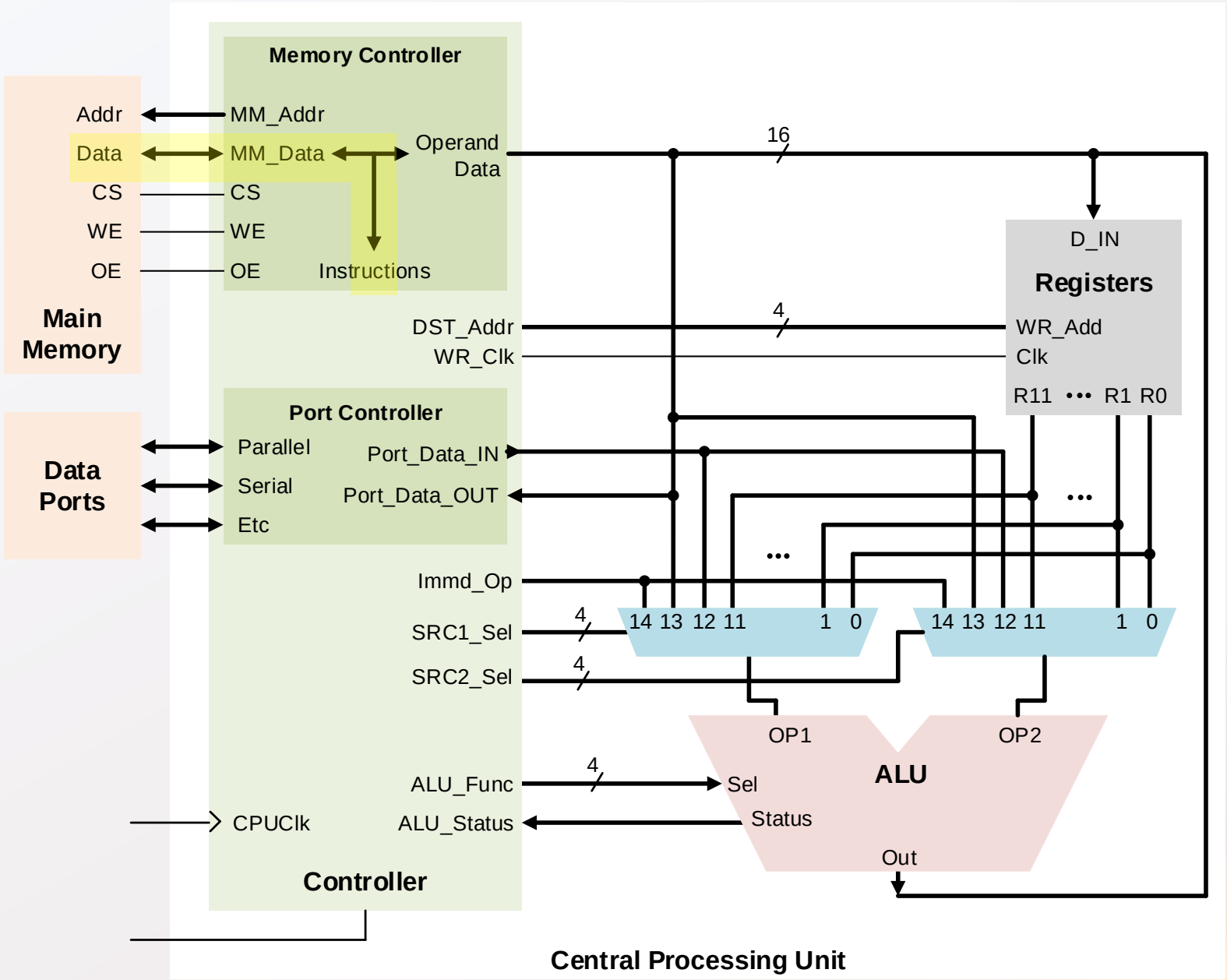
ALU Function	Code	Operation
ADD	0000	Out <= OP1+OP2
INC	0001	Out <= OP1+1
SUB	0010	Out <= OP1-OP2
DEC	0011	Out <= OP1-1
ADDI	0100	Out <= OP1+Immd
SUBI	0101	Out <= OP1-Immd
ASL	0110	Out <= OP1<<Val
ASR	0111	Out <= OP1>>Val
CLR	1000	Out <= 0
NOT	1001	Out <= !OP1
AND	1010	Out <= OP1&OP2
OR	1011	Out <= OP1|OP2
XOR	1100	Out <= OP1^OP2
XFER	1101	Out <= OP1
BZ	1110	Out <= OP1
JMP	1111	Out <= OP1



Every instruction has a unique 16-bit opcode. If main memory uses a 16-bit data bus, then one “read” can deliver one opcode to the controller. The CPU “fetches” each instruction from main memory immediately before it is executed.

“Programming” means storing lists of opcodes in main memory.

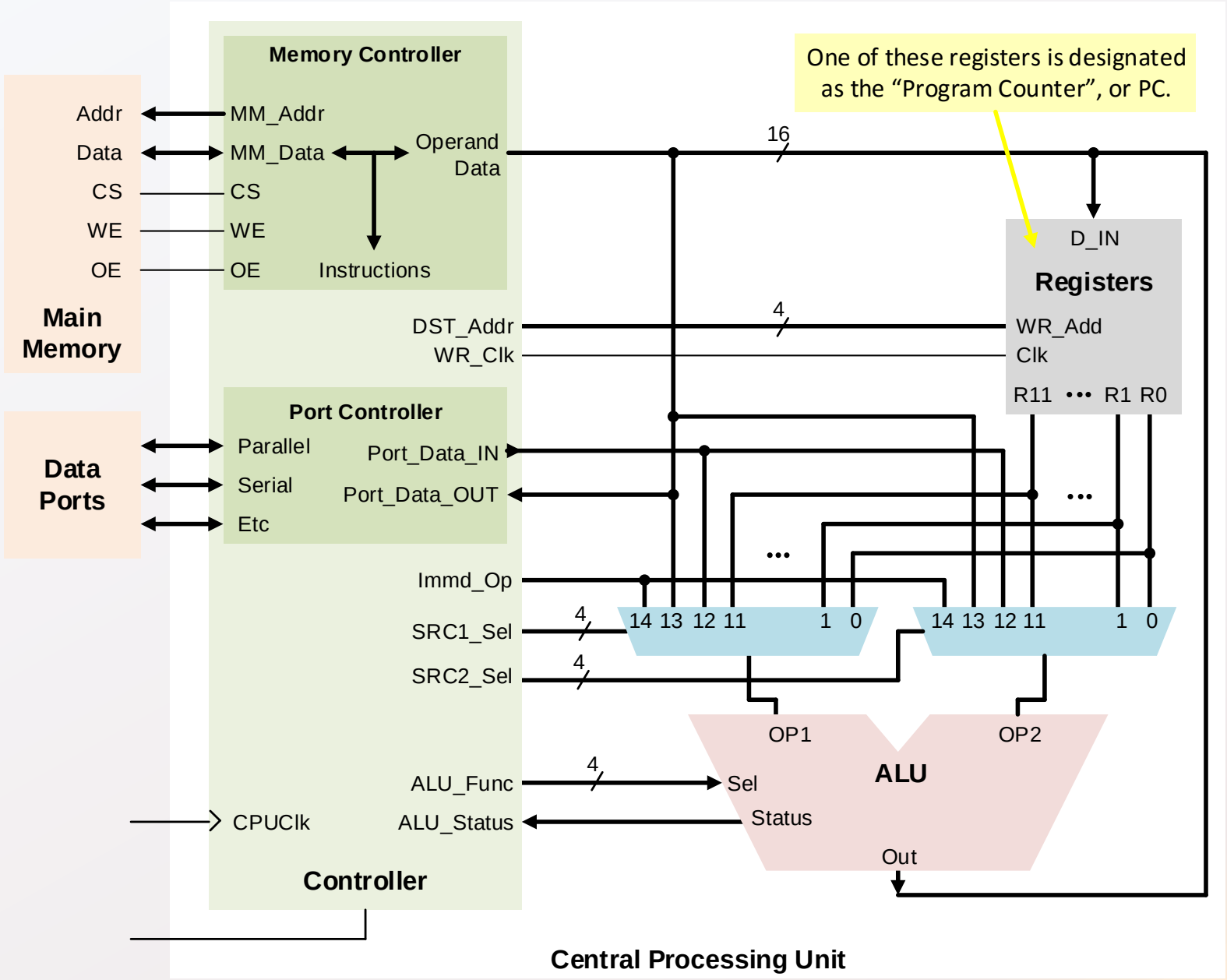
“Running” or “executing” the program means retrieving opcodes from memory one at a time, and executing them.



One of the registers is called the “Program Counter” (PC). The PC contains the main memory address of the opcode currently being executed.

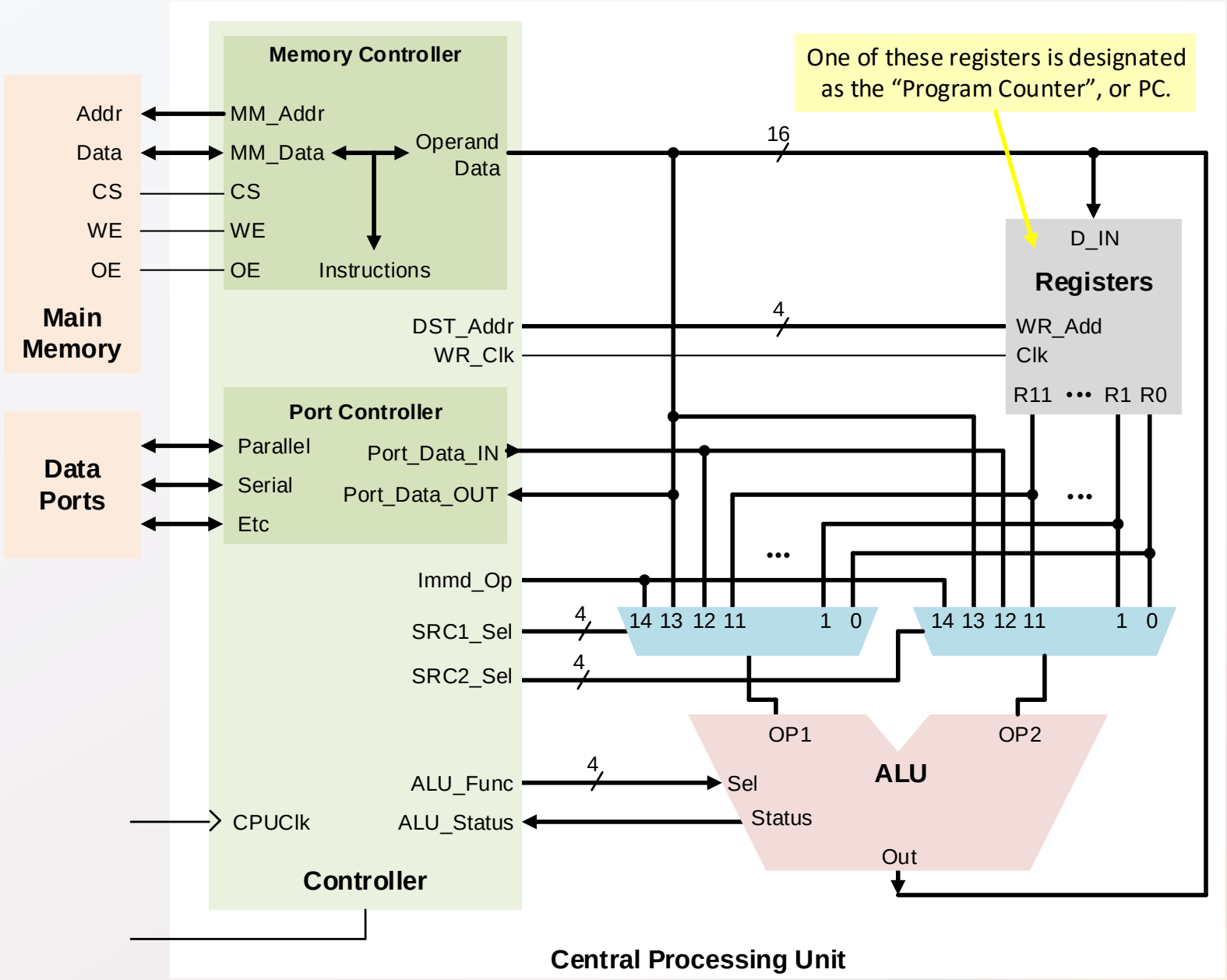
As the program runs, the PC is updated at the end of each instruction cycle to hold the address of the next opcode to be fetched and executed.

In a simple sequence, each “next” opcode is in the next sequential location in main memory. In this case, the controller simply increments the value in the PC.



How do you think the PC gets incremented?

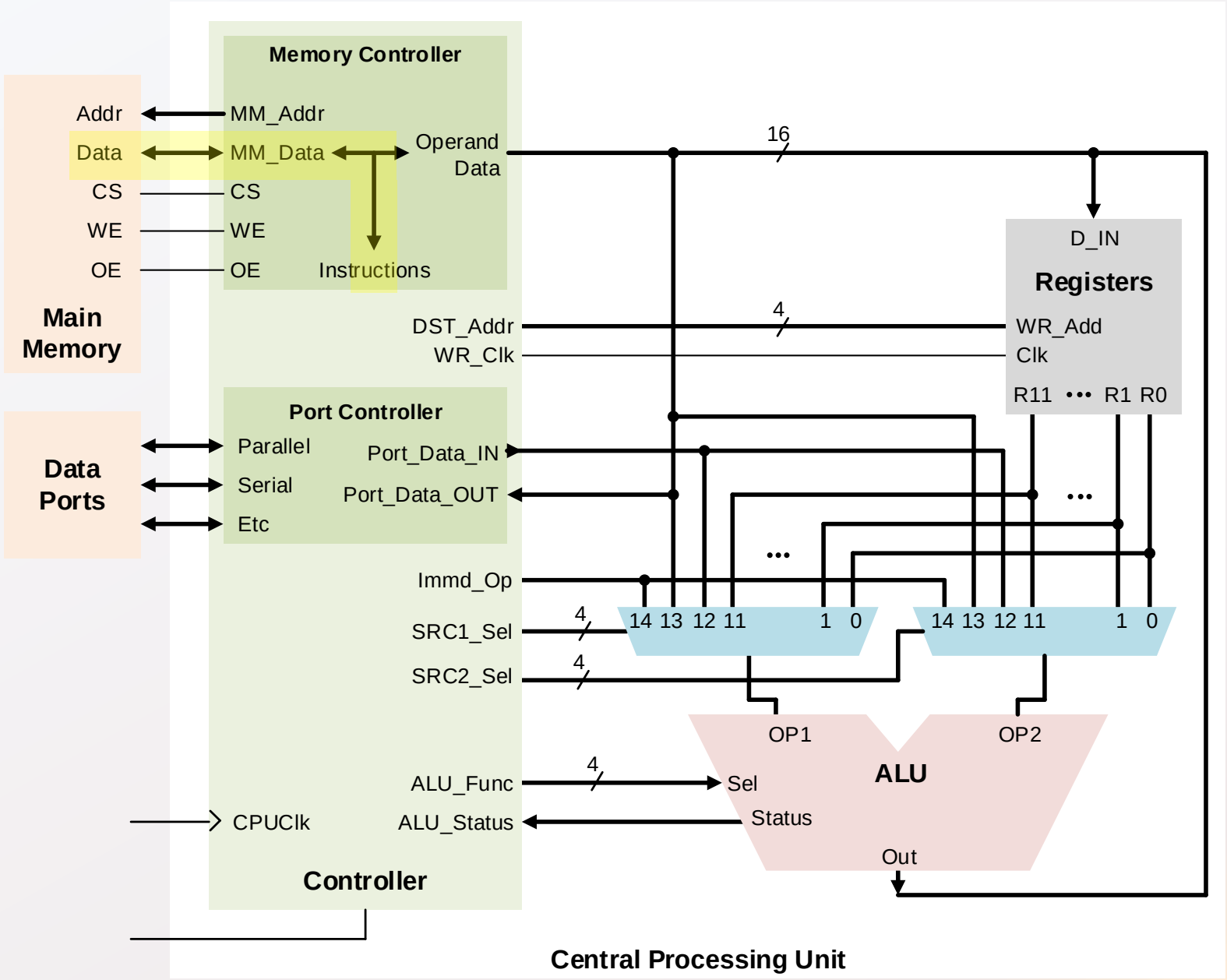
If only we had an ALU sitting around somewhere…



Two instructions, BZ and JMP, direct a new address value to be placed into the PC. Writing the PC with a new value means the next instruction can be fetched from anywhere in memory.

During program execution, the controller normally increments the PC. But if needed, a BZ or JMP instruction can direct the next instruction to come from somewhere else.

Why is this needed?

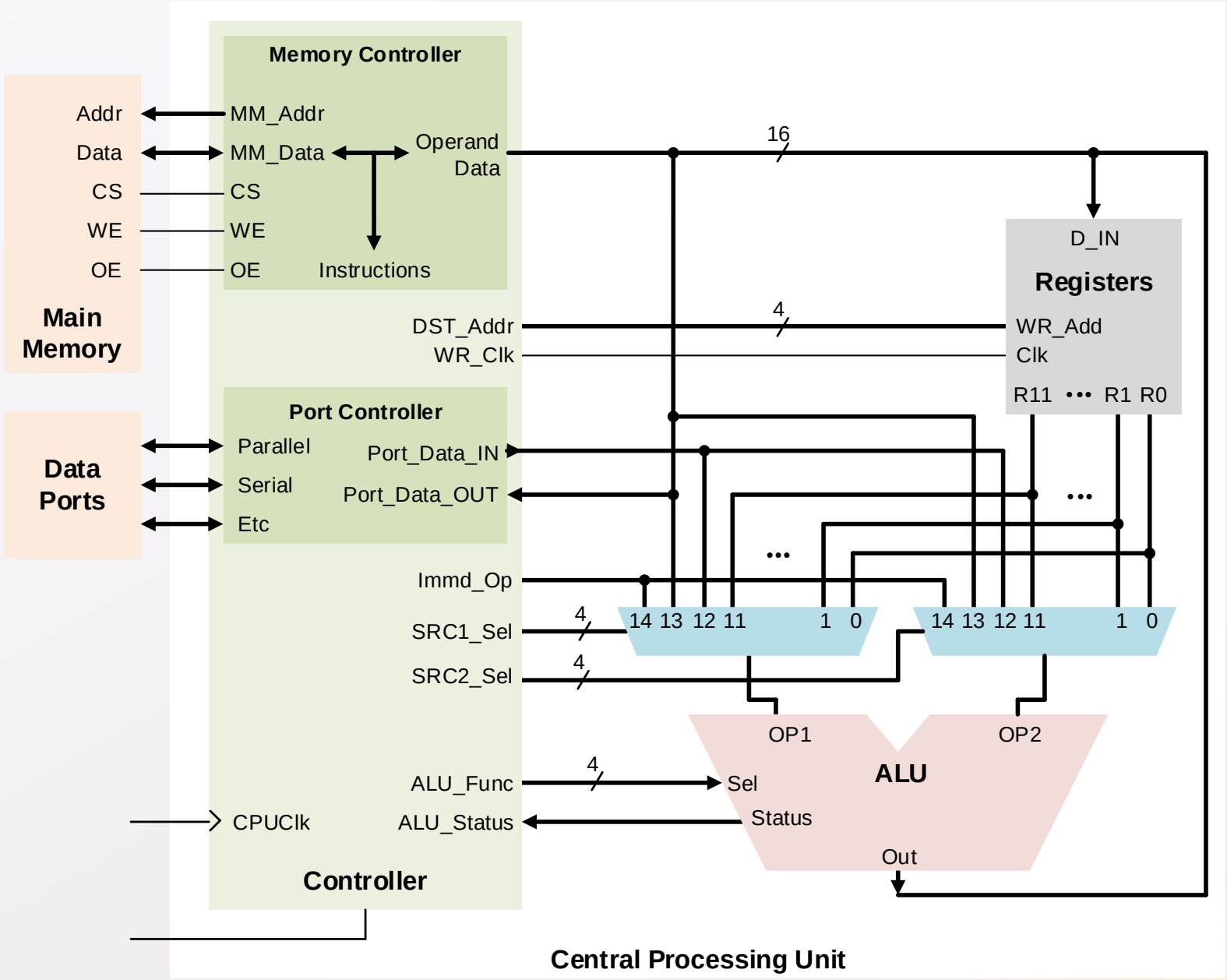


How to load data from main memory into a register?

- Set SRC1_Sel to 13;
- Set ALU to XFER (Load);
- Set DST_Addr to a register (ex: 4)
- Opcode 11010100----1101

(Note: SRC2_Sel is not used in this is a “single operand” instruction. The “----” means those bits are don’t cares).

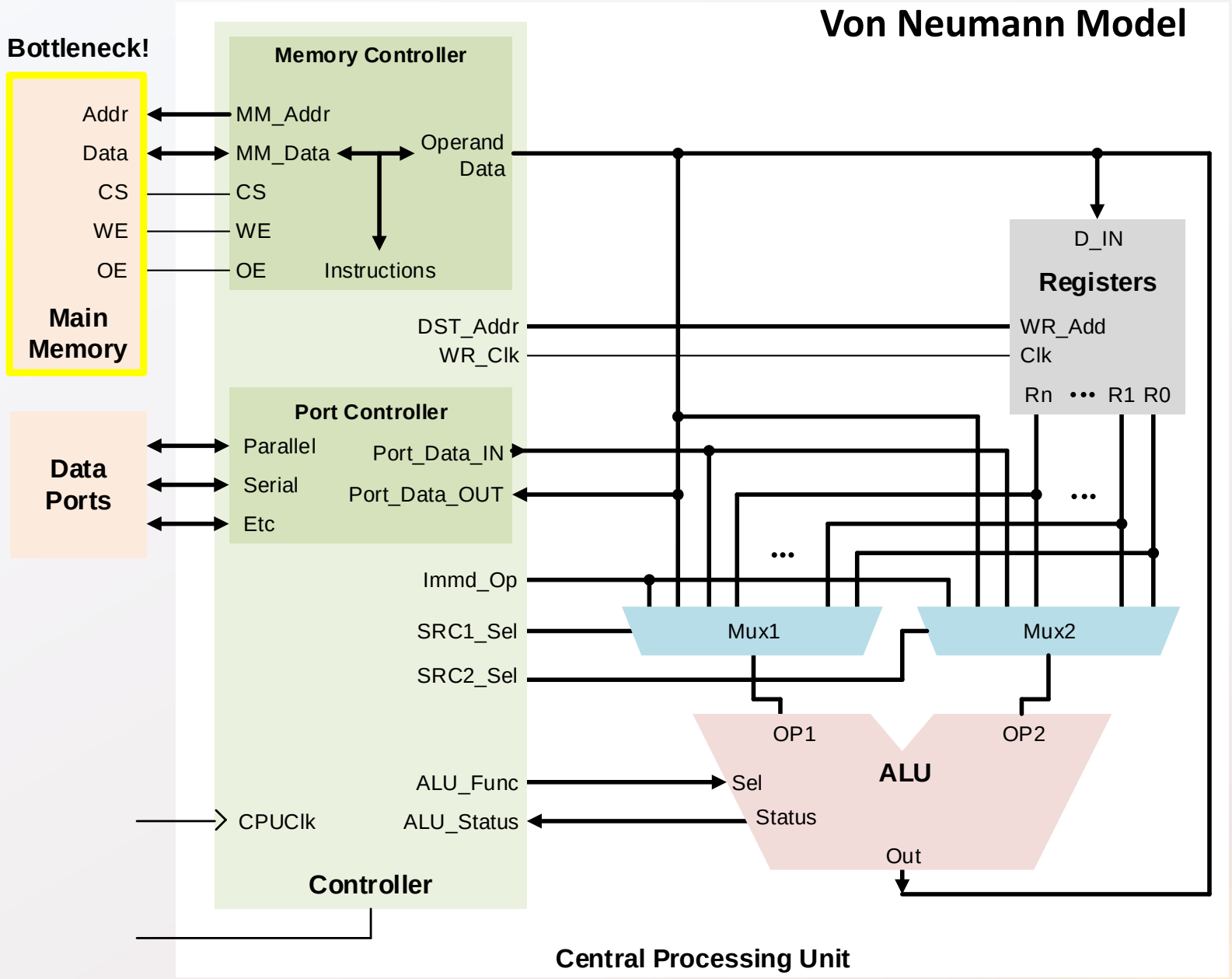
Main memory stores opcodes ***and*** data. This means the CPU must split access to main memory between opcode fetches, and data reads and writes.



This shared access to main memory creates a significant bottleneck that limits performance.

External processor pins are expensive, and designers try very hard to keep the number of processor pins at an absolute minimum.

Two different models have emerged.



Microprocessor Models

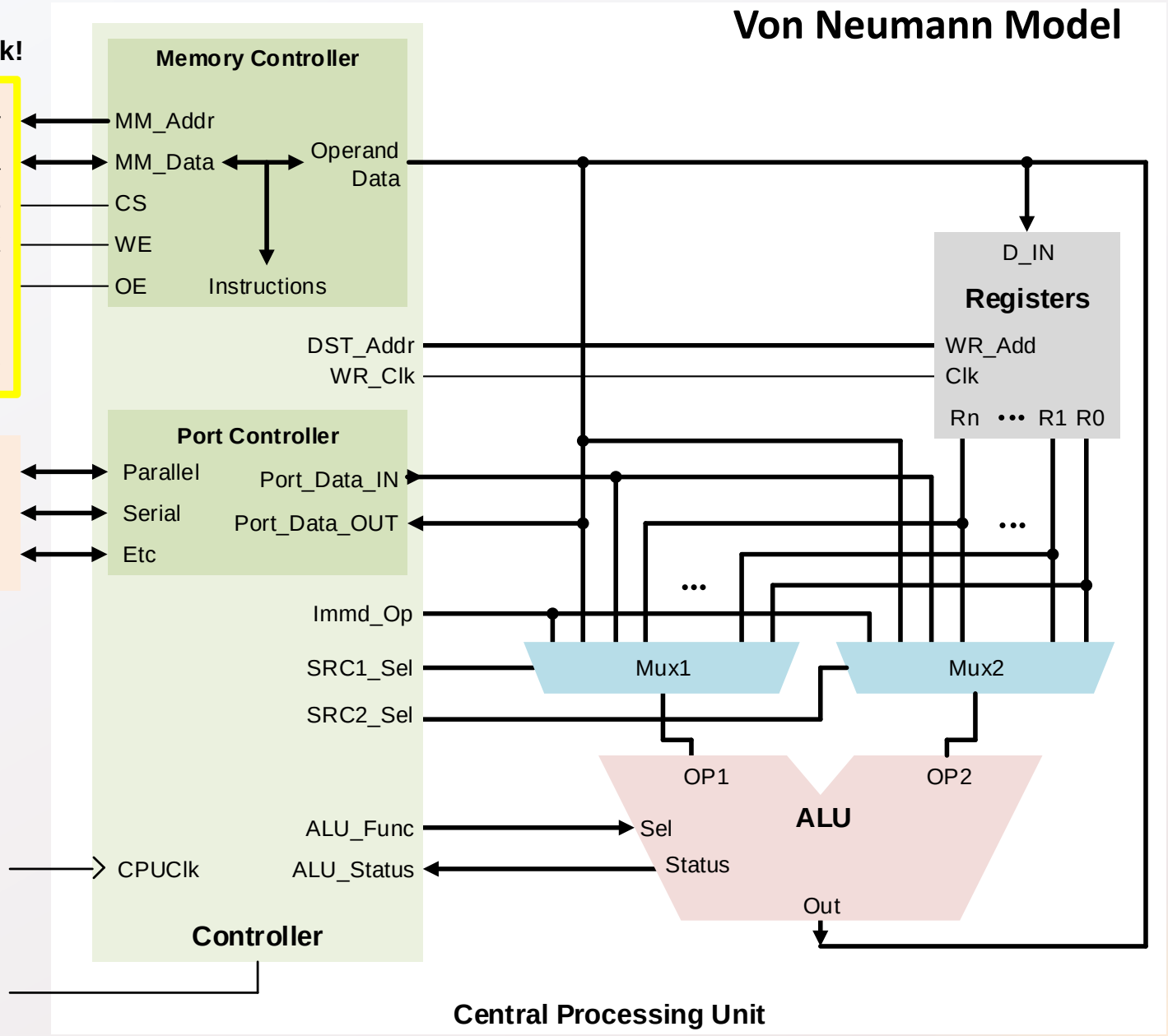
“Von Neumann” architecture

Shared memory for program and data – fundamental limitation

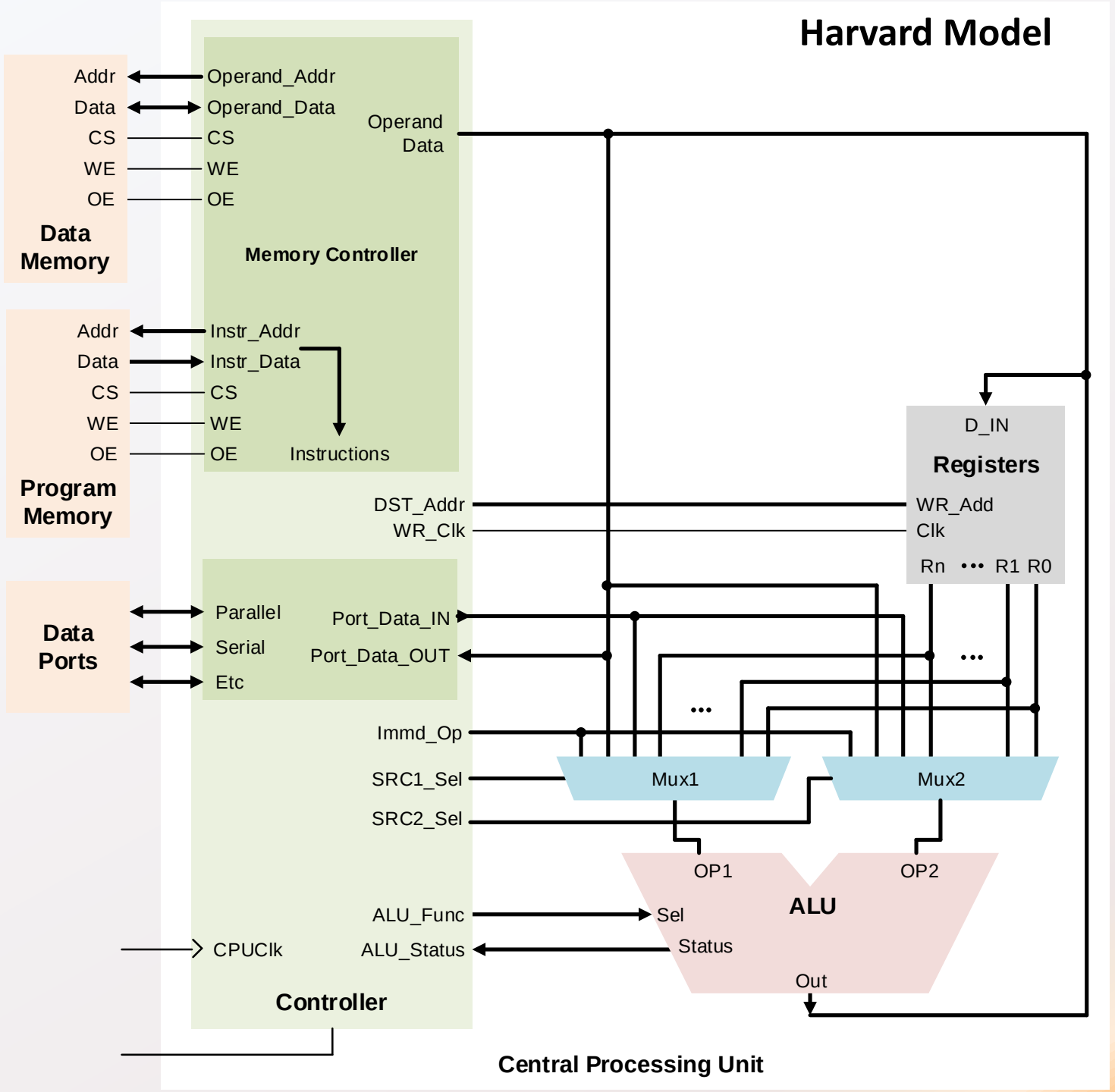
“Harvard” architecture

Separate memories for program and data (next slide)

Simpler, lower-cost processors use Von Neuman model; higher performance processors use Harvard.



In the Harvard model, opcodes come from program memory, and data from data memory.

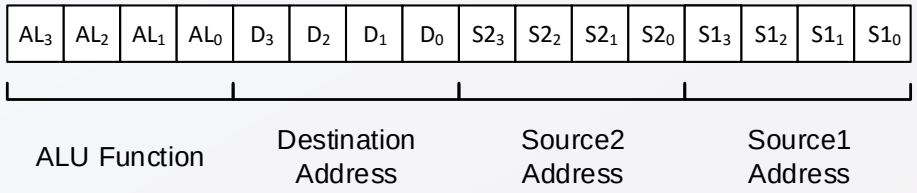


Earlier, we defined opcode

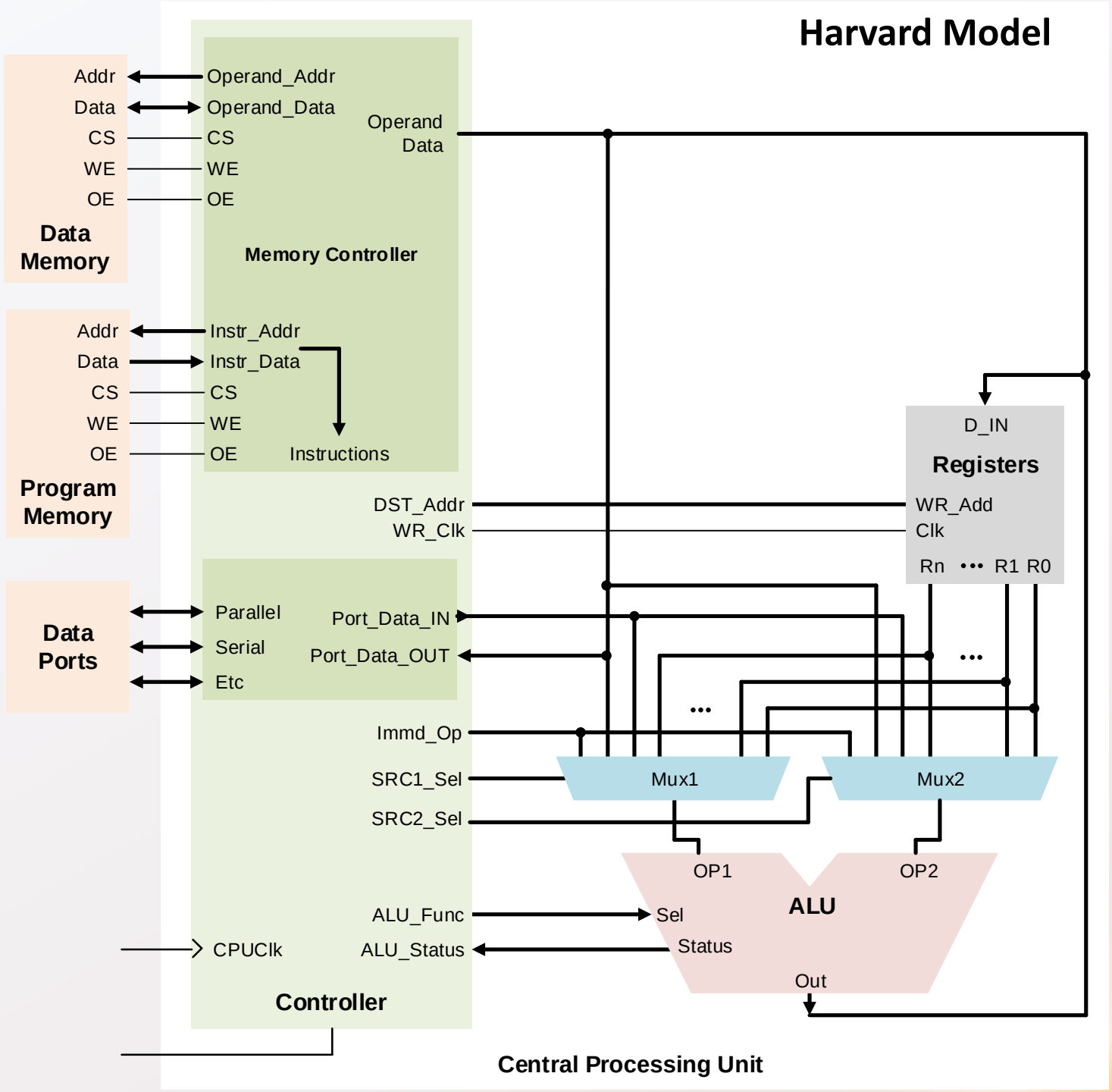
11010100----1101

as an instruction to move data from main memory into register 4.

ALU Function	Code	Operation
XFER	1101	Out <= OP1

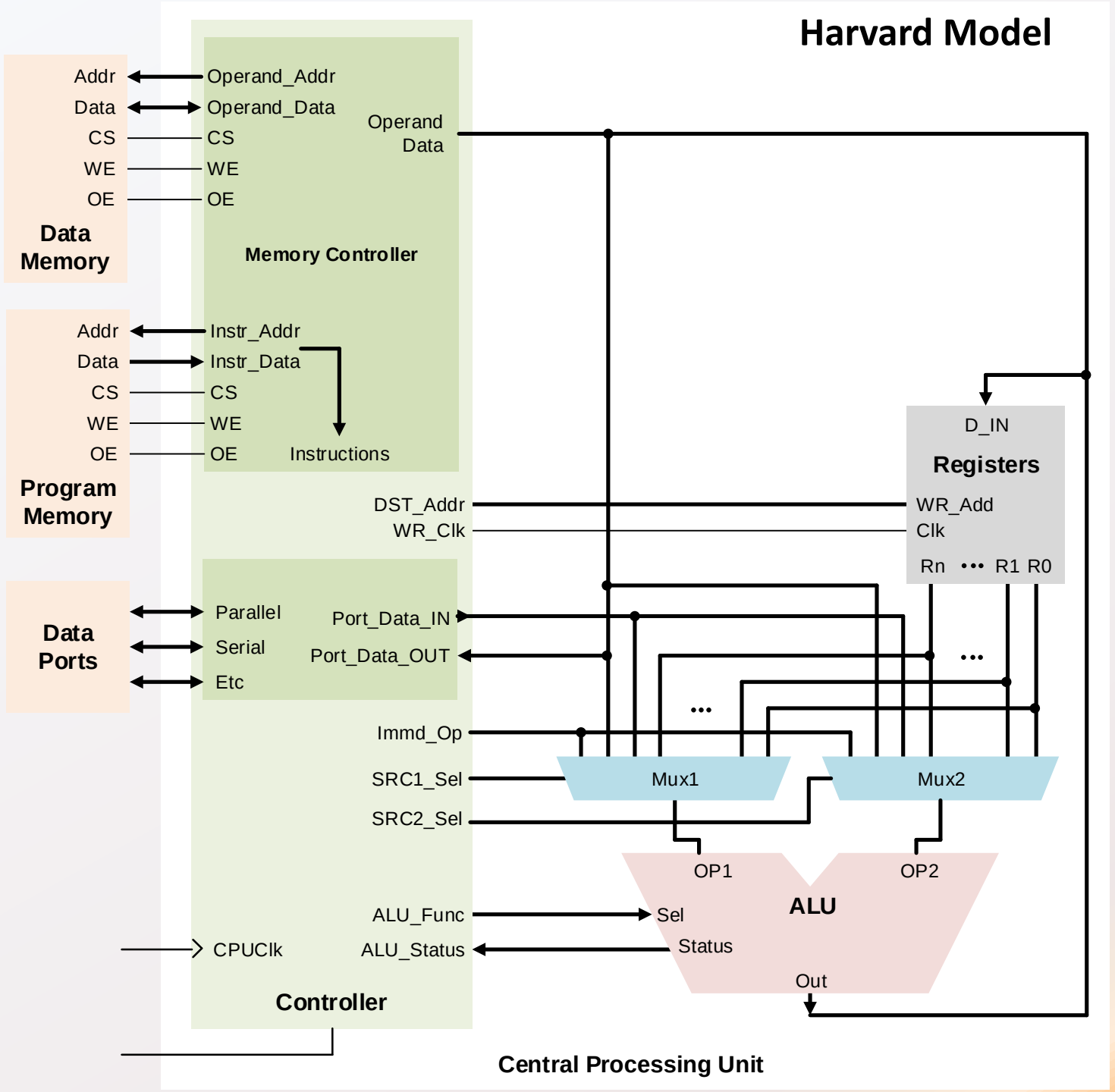


Where does the data memory address come from? There's no room in the opcode…



In our example, if an opcode moves memory data to or from the CPU, the data element immediately following the opcode in program memory is the data address.

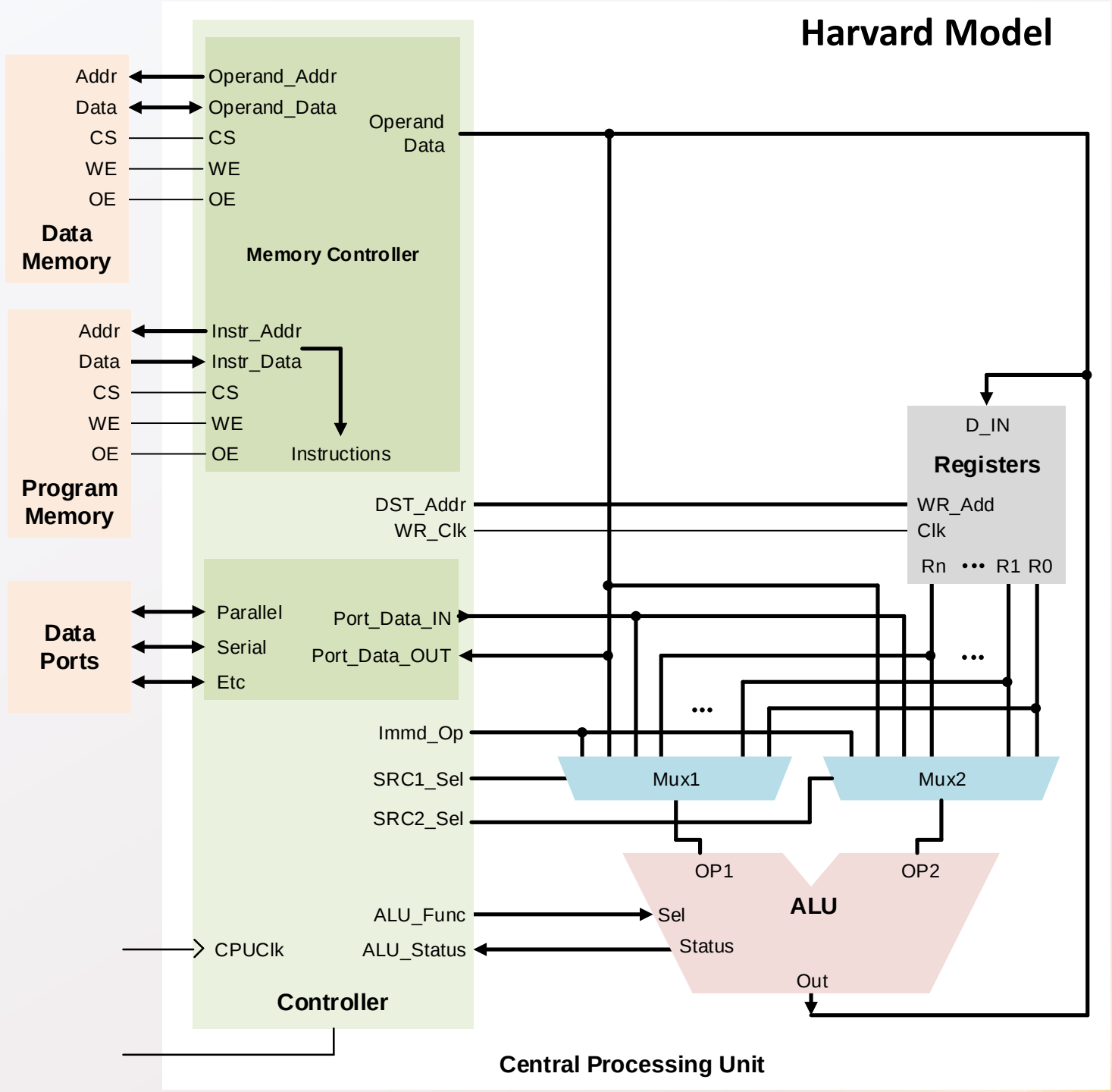
The controller must recognize the XFER opcode, and that the source or destination operand is in main memory. Then it must access memory to read or write the data at the address provided before moving on to the next instruction.



Some processers require that memory addresses must be transferred to an “address register” before the memory can be accessed.

Address registers are like other registers, but their outputs are directed to the memory controller as well as the ALU.

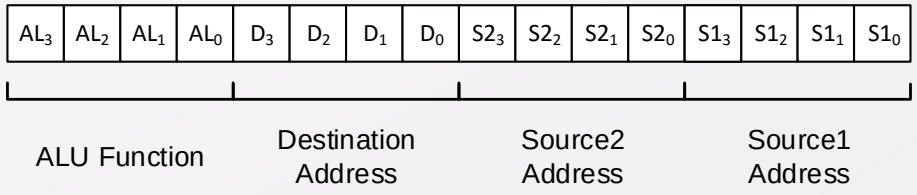
It takes more time to fetch an operand from main memory than from a register. The controller must “pause” operations to wait for memory operands.

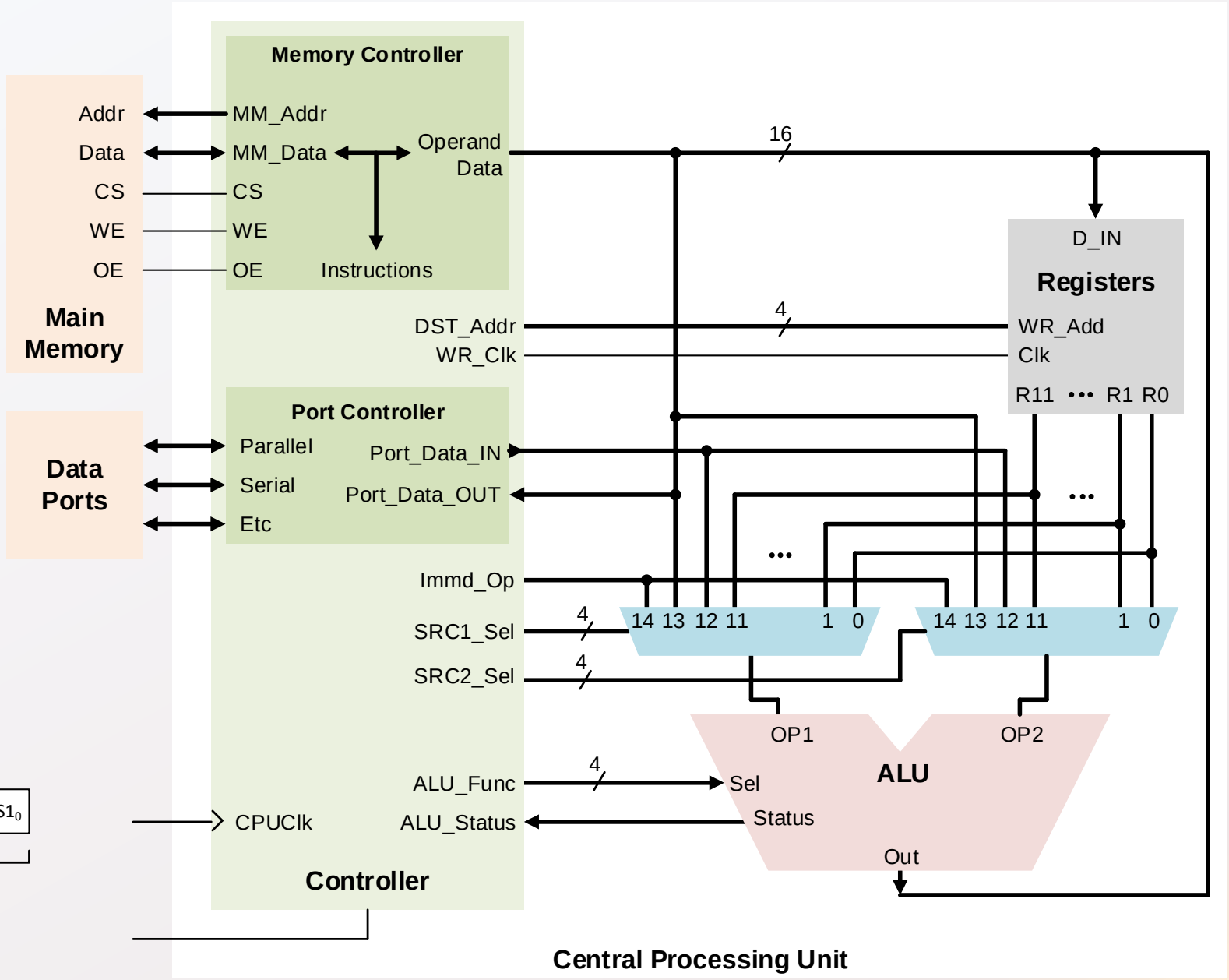


Three more observations…

The ASL and ASR instructions also use a single operand, but they need a 4-bit value “Val” to define the number of bits to shift. For these instructions, the otherwise unused four S2 bits are “overloaded”, and used to define “Val”.

ALU Function	Code	Operation
ASL	0110	Out <= OP1<<Val
ASR	0111	Out <= OP1>>Val

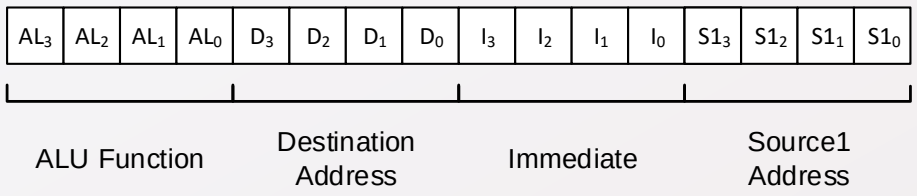


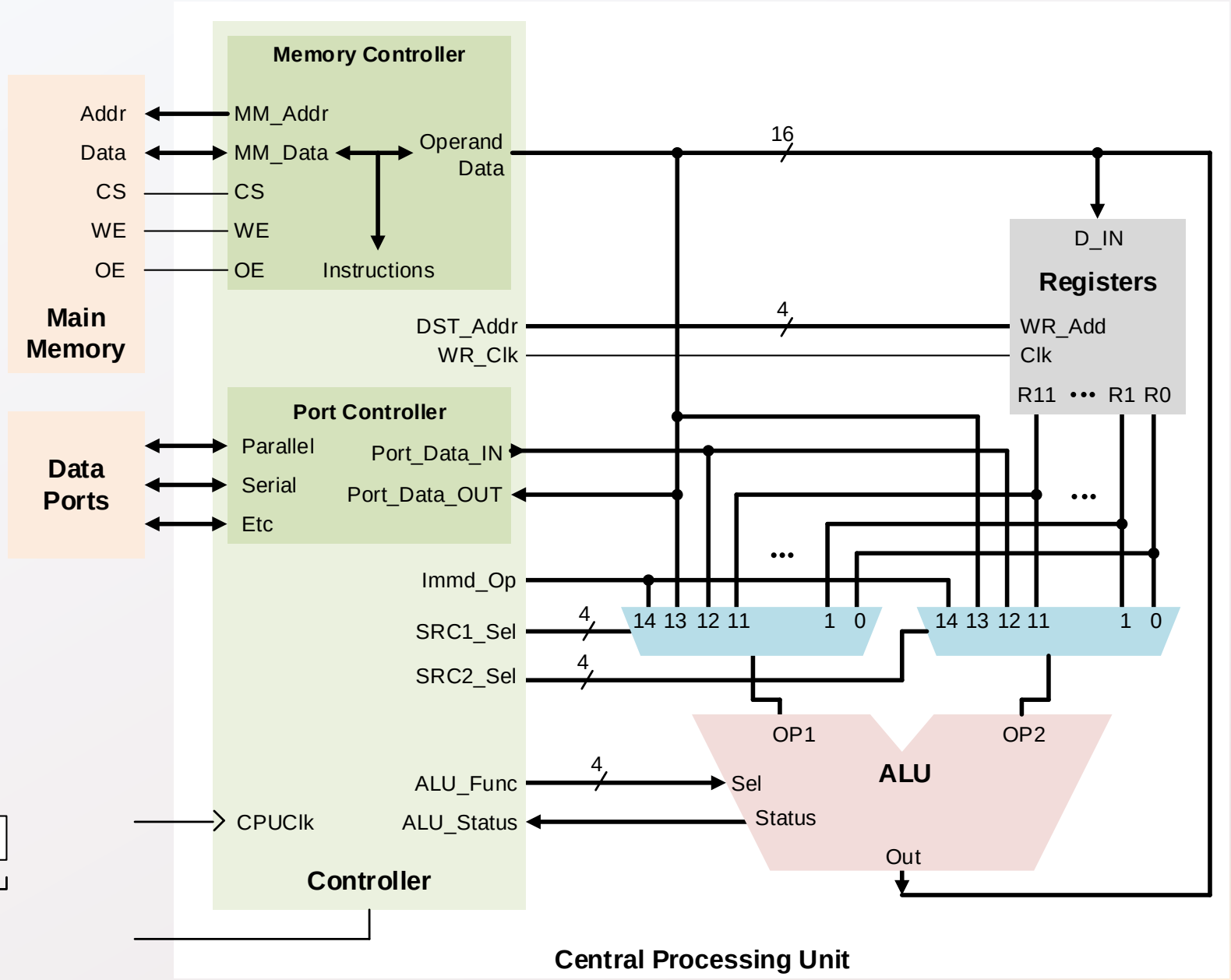


When operands (like Val) are in the opcode itself, they are called “immediate” operands.

Val is an immediate.

ALU Function	Code	Operation
ASL	0110	Out <= OP1<<Val
ASR	0111	Out <= OP1>>Val

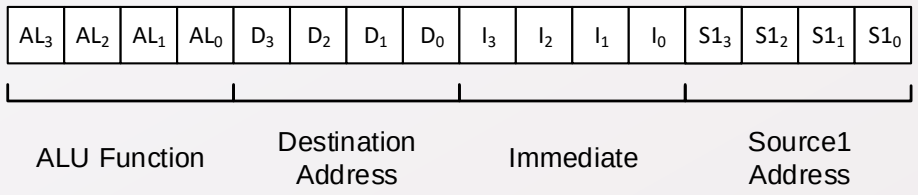


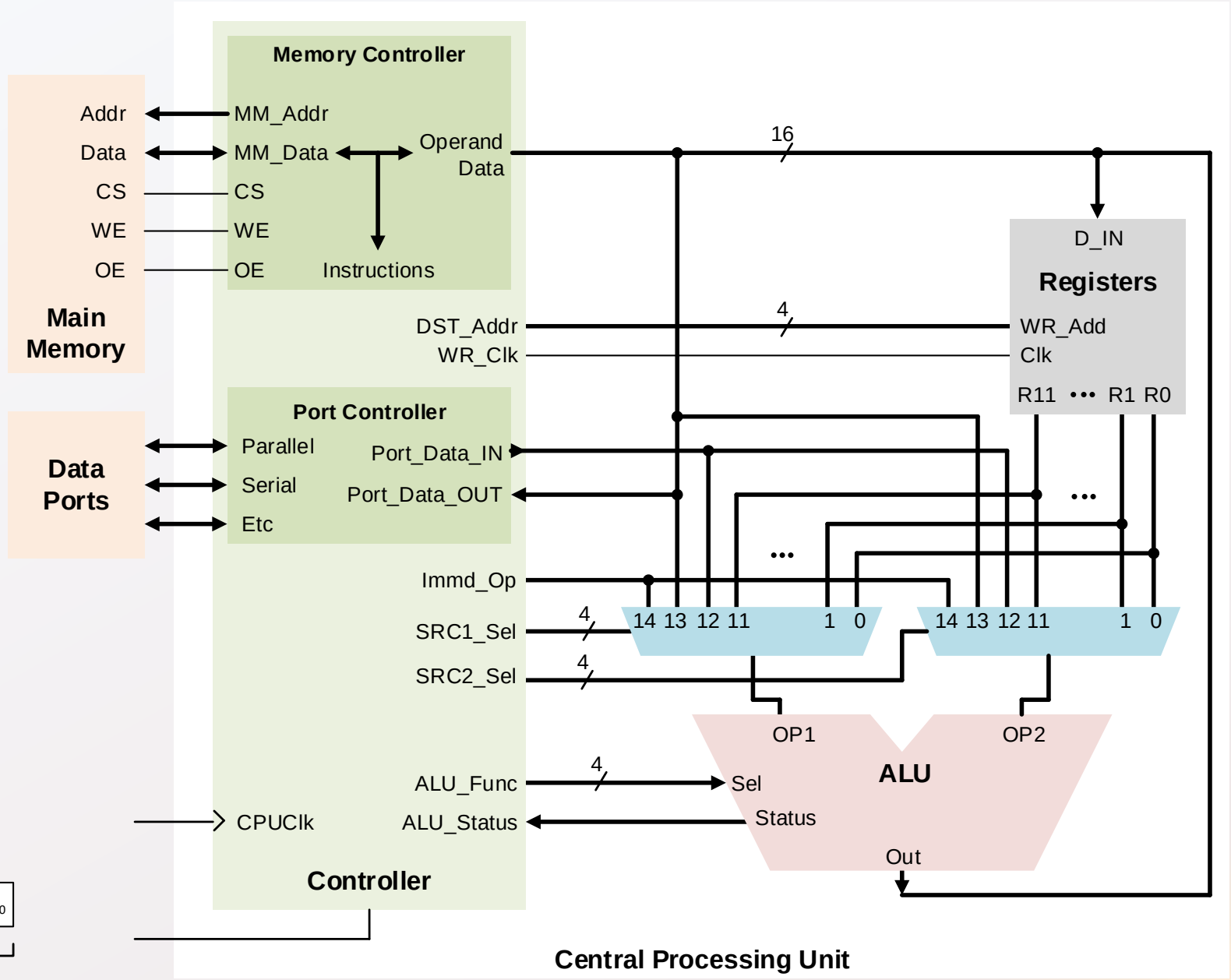


The ADDI and SUBI instructions add or subtract an immediate operand from operand1.

Immediate values are stored in program memory along with the instructions (typically, they use extra bits in the opcode, or the next memory location just after the instruction). In most CPUs, access to immediate operands is quicker than access to other memory.

ALU Function	Code	Operation
ADDI	0100	Out <= OP1+Immd
SUBI	0101	Out <= OP1-Immd

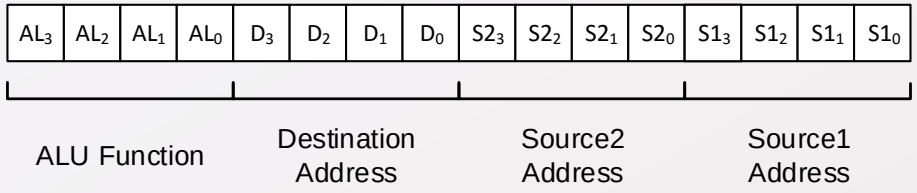


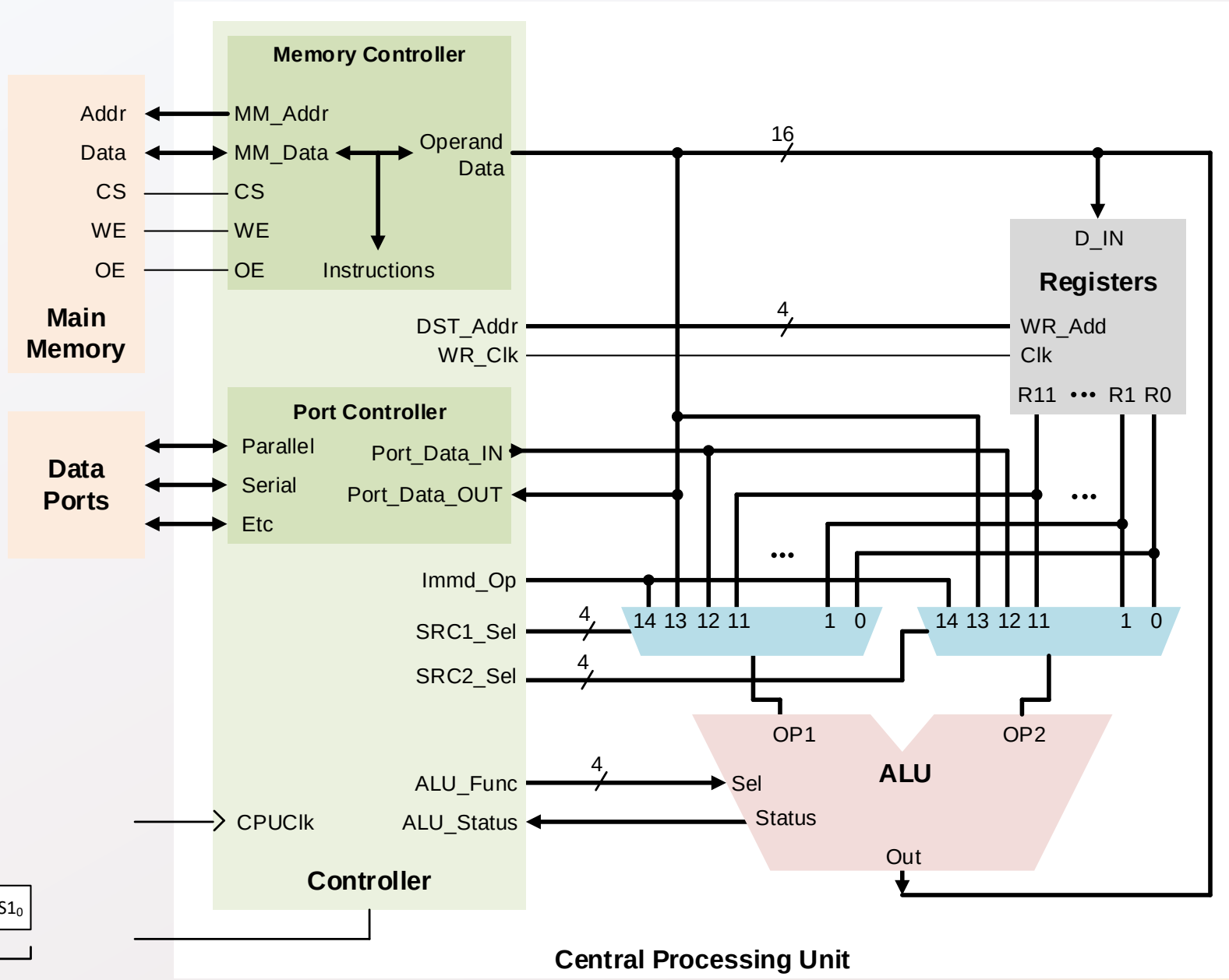


And…

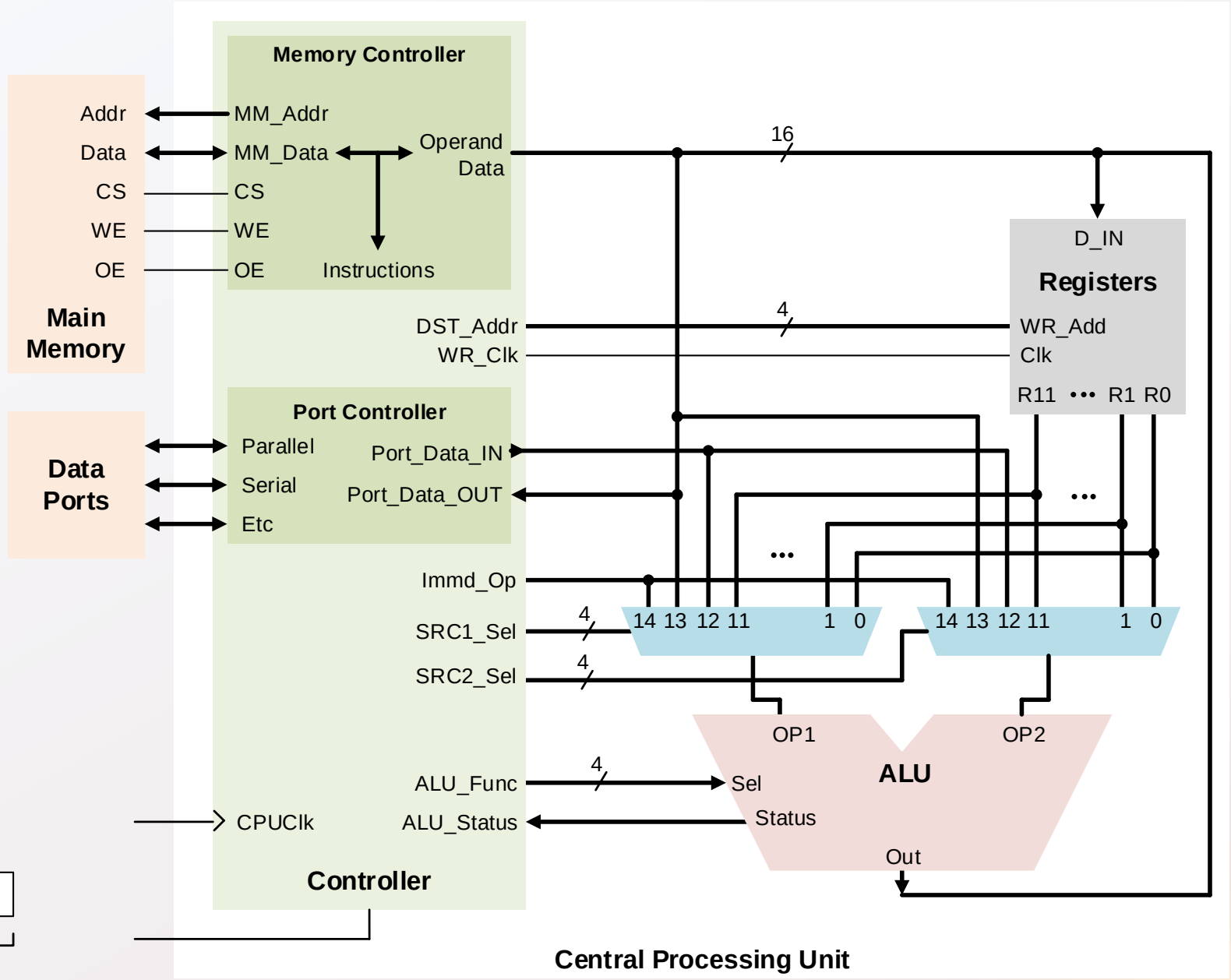
The BZ and JMP instructions do not use the ALU like all other instructions do. Instead, they “redirect” program flow, by causing the next fetch to come from a new, out of sequence address. That address is stored in program memory in the next memory location after the instruction.

ALU Function	Code	Operation
BZ	1110	Out <= OP1
JMP	1111	Out <= OP1

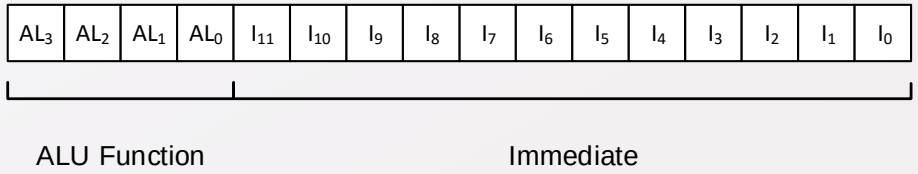




Since JMP or BZ don't need any operands, we could use all 12 bits as an immediate. That would let us store a 12-bit number. Is that enough?



ALU Function	Code	Operation
BZ	1110	Out <= OP1
JMP	1111	Out <= OP1

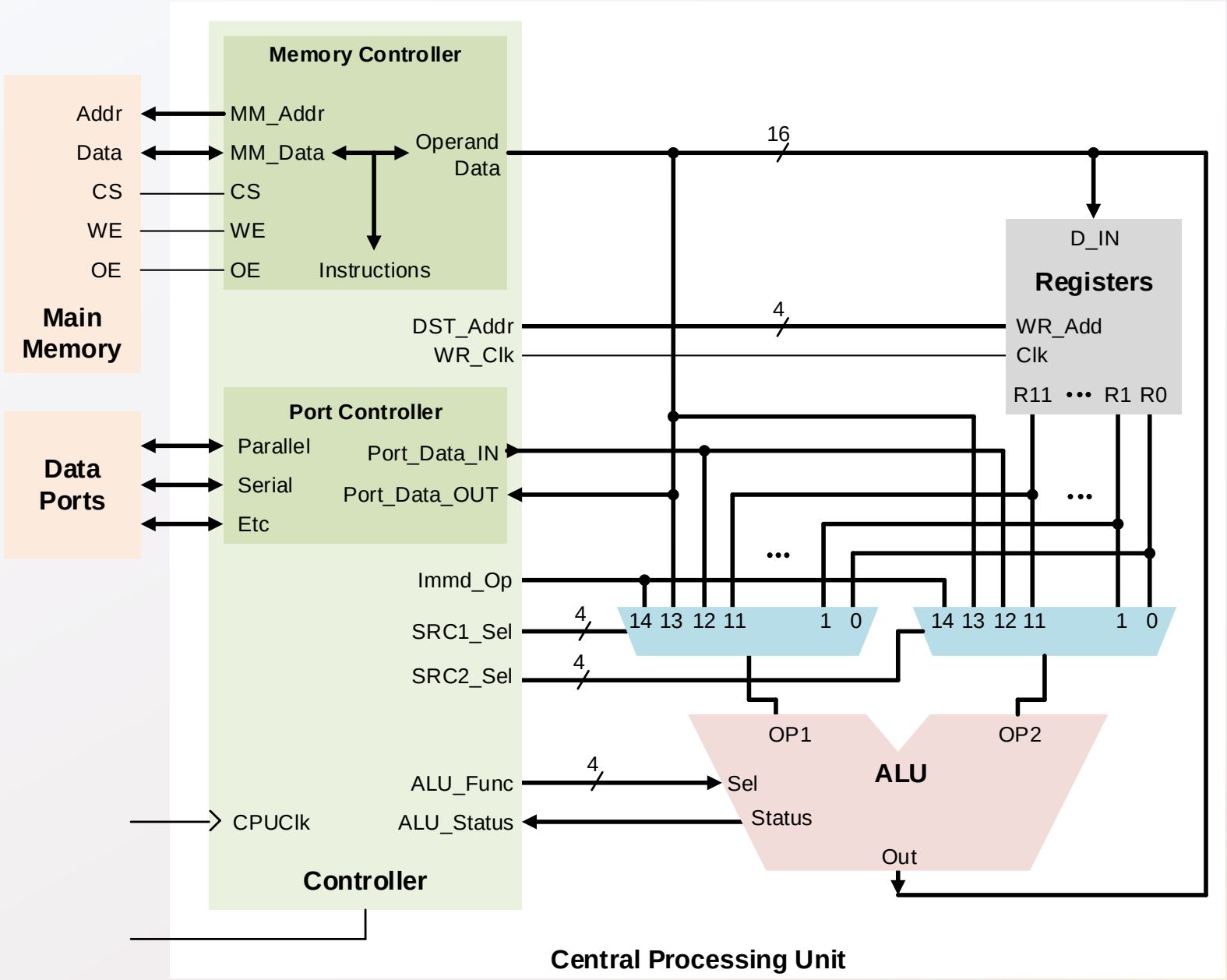


Some processors require that all ALU source operands come from registers – no ALU operations can directly access operands in main memory.

In these processors, operands must first be transferred to a register, and the result must be stored in a register. Then the result can be transferred back to memory.

This is called a “load-store” architecture.

Can you think of advantages to this architecture?

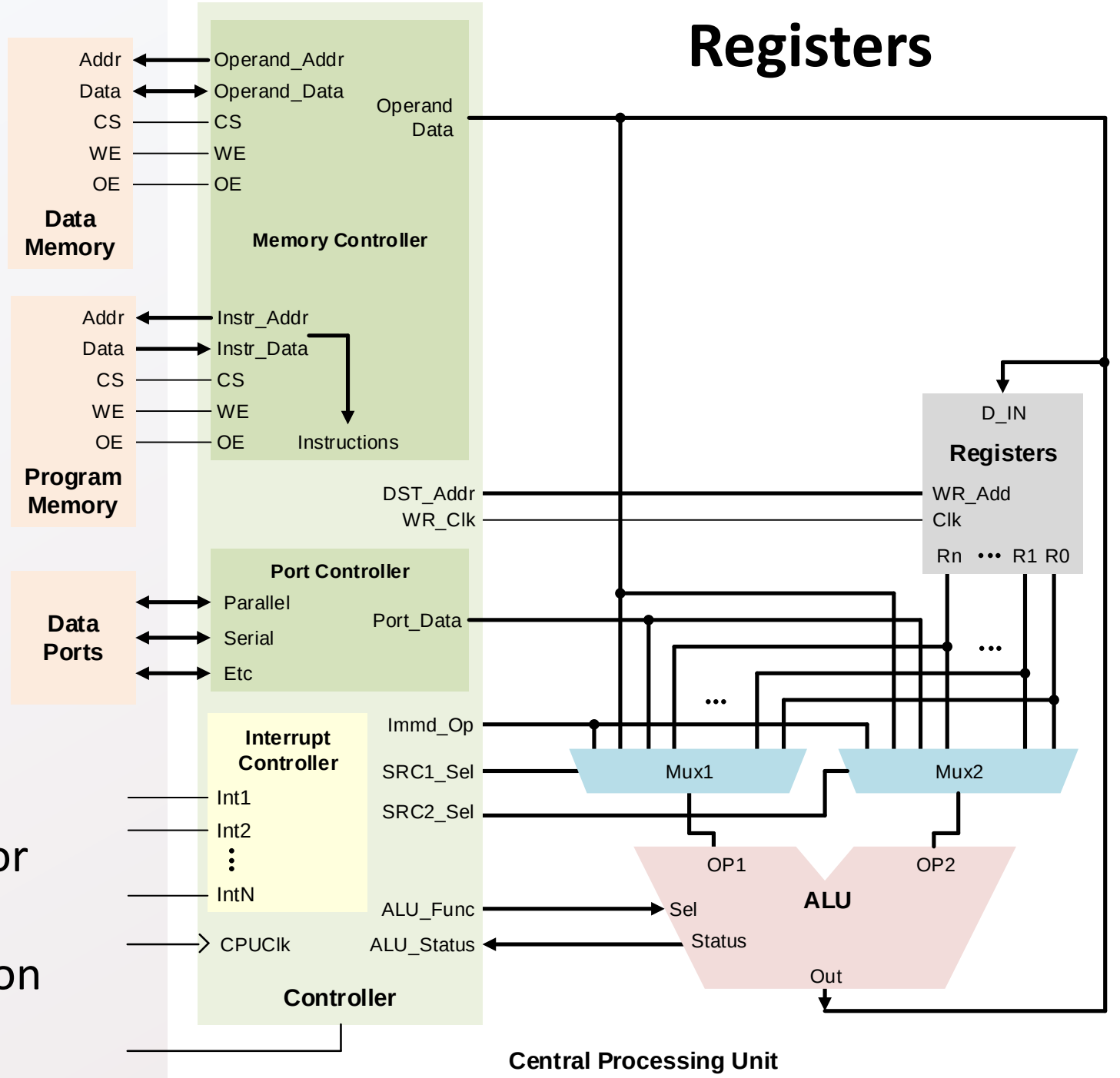


Registers

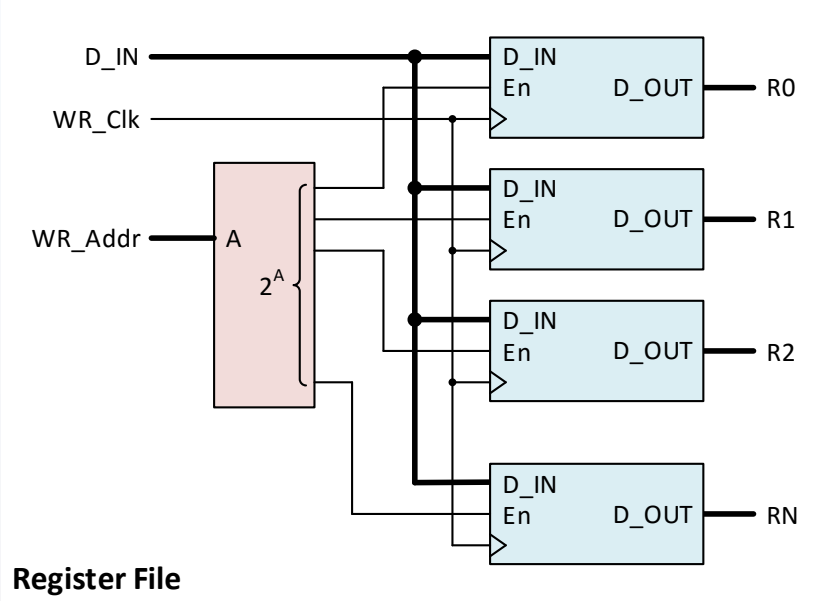
Registers are high-speed, unshared memories that are immediately available to the CPU. There are few of them, and they are used for ALU operand storage and a few critical CPU operations.

Registers are often organized as multi-port register files, with two read ports and one write port that can all be active simultaneously.

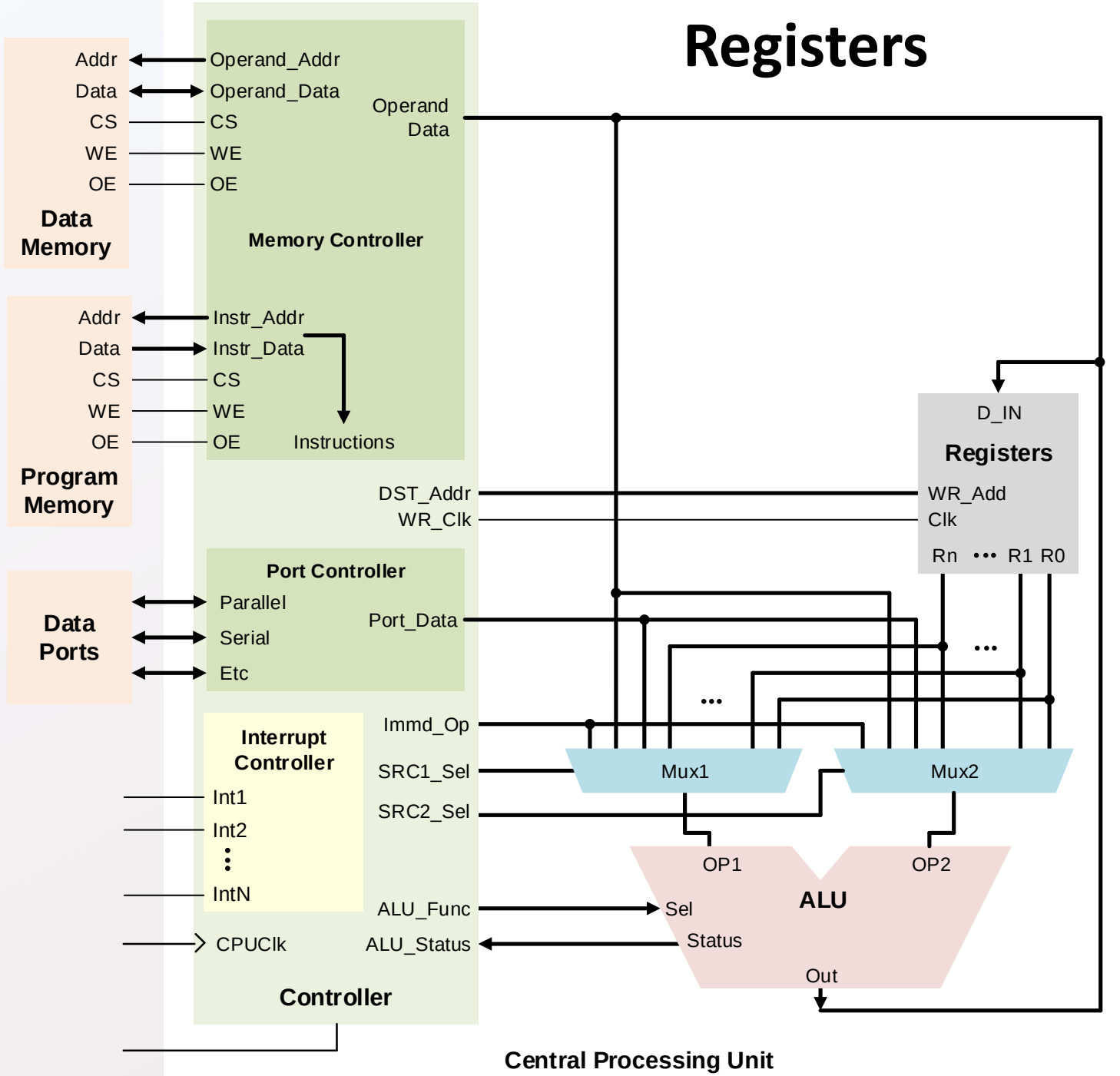
Many processors use a load-store architecture, where data is **loaded** from larger, slower memories into registers for use by the ALU. ALU output is initially stored in a register, and a later instruction can **store** it back into main memory.



Note that in our example CPU, the registers are directly accessible (through muxes) by the controller/ALU.



This is the most efficient case. Some processors use “memory mapped” registers that are accessed through a shared bus. These are slower, and generally the ALU cannot access them directly.



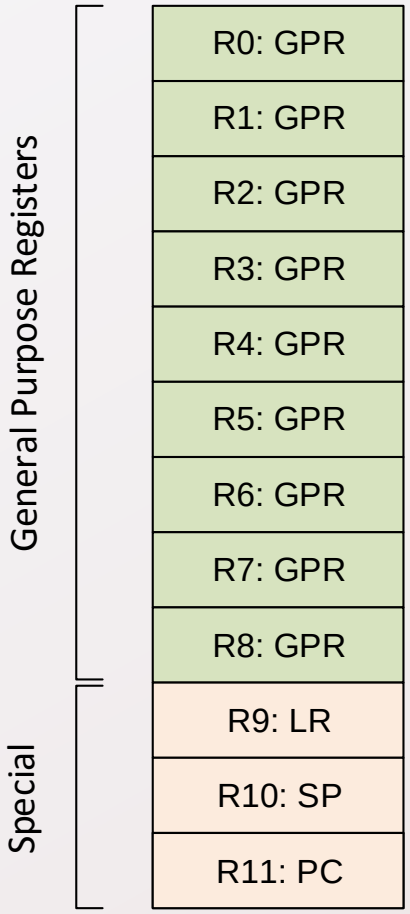
Registers

Our example CPU has 12 registers. This is a typical number – some processors may have 32, some may have just a few. Most registers are “general purpose”, and are used to store temporary/working data for the ALU.

Some registers are special purpose, and store critical data the CPU needs. In this example, R11 is the program counter. If R11 is written to a new value, the processor will fetch its next instruction from that address (this is how jumps occur).

R10 is the Stack Pointer. The stack is a memory array that stores “context” information when subroutines or interrupts are processed, and the value in R10 shows the current “top” of the stack. More on stacks later.

R9 is the Link Register. The link registers stores the return address that is used when a function call completes. More on this later as well.



Registers

All processors use some form of registers. Some have more classes of registers than the two in our example.

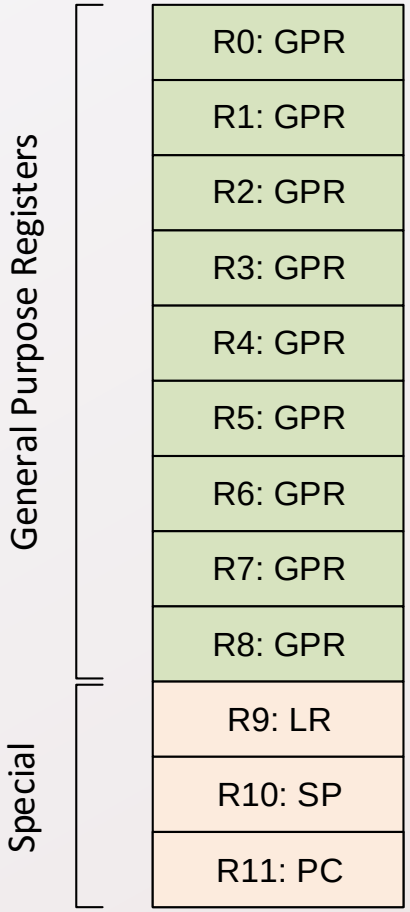
For example, many Intel processors use address registers like “base index” or EBX, “destination index” or EDI, and “source index” or ESI to store the memory addresses of variables. Atmel processors also use address registers.

Some processors offer a collection of floating point registers to hold data values used in floating point operations.

Some processors use port registers to hold port data values, as well as configuration bits like tri-state control, latching of data, pin direction (in or out), etc.

Some processors have an instruction register that holds the instruction code for the currently executing instruction.

Most processors use a status register that holds information about the currently executing instruction… more on status shortly.



In addition to the “out” signals, ALUs also produce “status bits” to provide information about the most recent ALU operation.

S: Sign bit; set if a negative number was the last ALU output

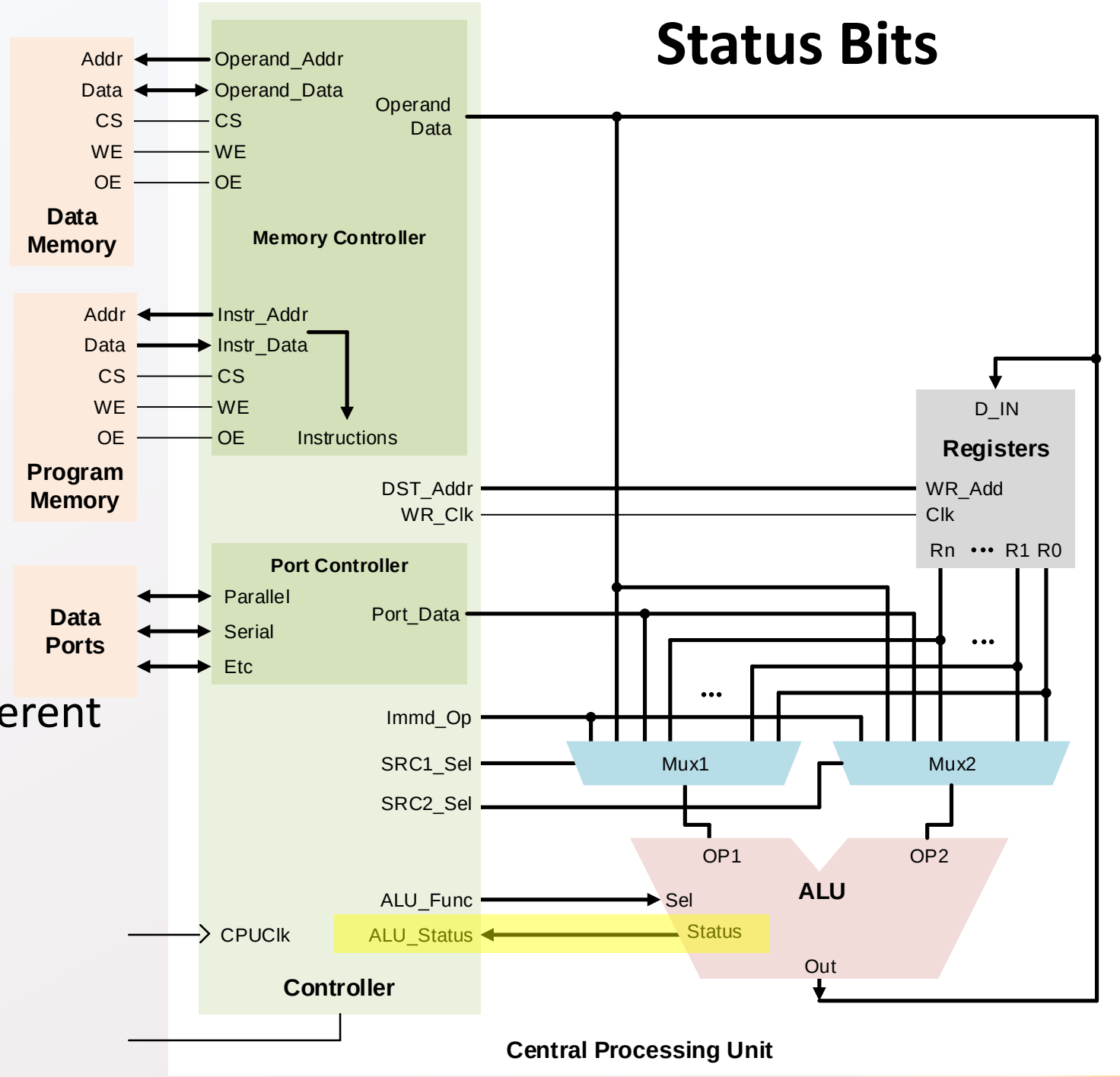
V: Overflow bit; set if 2’s compliment operation resulted in overflow

Z: Zero bit; set if ALU output is zero

C: Carry bit; set if last operation produced a carry out

Different CPUs/ALUs will produce different status bit – this is more or less the minimum set.

Status bits are set by the previous instruction, and they remain current throughout the instruction cycle.



For certain instructions, the controller state machine bases its state flow on the status bits. For example, the BZ instruction directs the controller to "Branch on Zero" to a new address instead of just incrementing the PC (and recall that address is stored in the next location in program memory, immediately after the opcode). If the most recent ALU operation resulted in a "0" output, the Z bit would be set throughout the current instruction execution cycle.

The controller state machine will load the PC with a new address if code "1110" and "Z = 1".

ALU Function	Code	Operation
BZ	1110	Out <= OP1

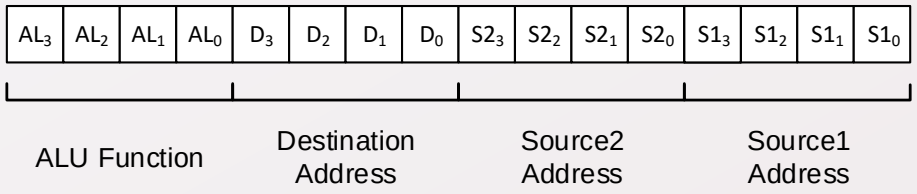
Most processors have instructions that explicitly or implicitly check status bits. Explicit instructions might include CMP (compare) or TST (Test).

Implicit instructions might include "Branch on Carry", or "Branch on Negative".

ALU Function	Code	Operation
ADD	0000	Out <= OP1+OP2
INC	0001	Out <= OP1+1
SUB	0010	Out <= OP1-OP2
DEC	0011	Out <= OP1-1
ADDI	0100	Out <= OP1+Immd
SUBI	0101	Out <= OP1-Immd
ASL	0110	Out <= OP1<<Val
ASR	0111	Out <= OP1>>Val
CLR	1000	Out <= 0
NOT	1001	Out <= !OP1
AND	1010	Out <= OP1&OP2
OR	1011	Out <= OP1|OP2
XOR	1100	Out <= OP1^OP2
XFER	1101	Out <= OP1
BZ	1110	Out <= OP1
JMP	1111	Out <= OP1

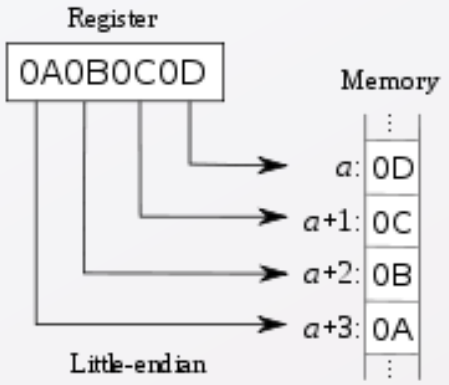
The last example shows that opcodes configure the ALU and provide inputs to the controller state machine.

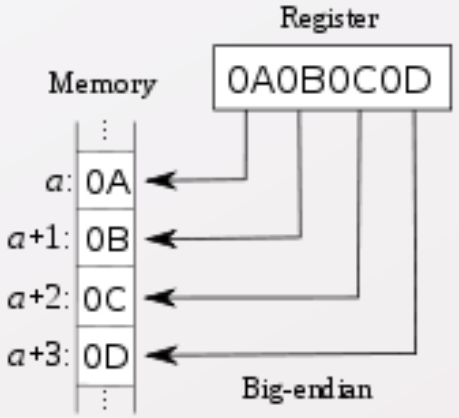
Another example - the last three rows in our ALU function table all assign Out <= OP1. They configure the ALU in the same way, and may seem redundant. But they also direct the controller state machine to take different actions. The BZ instruction is one example – the state machine would only check the Z bit if BZ were the active instruction.



The XFER function is another example. Any source can be transferred to any destination. For example, R2 can be transferred to R10, or R3 can be transferred to a Port. Transfers to main memory (or a port) will take longer, because memory cycles take more time. The controller state machine must generate memory control signals, and allow for the extra time.

Big Endian vs. Little Endian





RISC: Reduced Instruction Set Computer

Simple, fast instructions that complete in one cycle.

Operands are all the same size (ex: 32 bits).

Operands must use registers.

Example: R3 = R2 + R1;

Load-Store architecture

More explicit steps; more source code

Simpler, faster, less expensive ICs

ARM, MIPS, PIC, AVR

CISC: Complex Instruction Set Computer

Complex instructions that can take several cycles to complete.

Operands have variable sizes.

Operands can come from memory.

Example: data2 = data1 + R1;

This would “unroll” into: load operand data1 into a register; add it to R1; place result in a temporary holding register; write register to memory at data2.

Fewer explicit steps; less source code

More complex, more expensive hardware

Intel