Lecture 2

COP3402 FALL 2015 – DR. MATTHEW GERBER – 8/26/2015

FROM EURIPIDES MONTAGNE, FALL 2014

Tonight

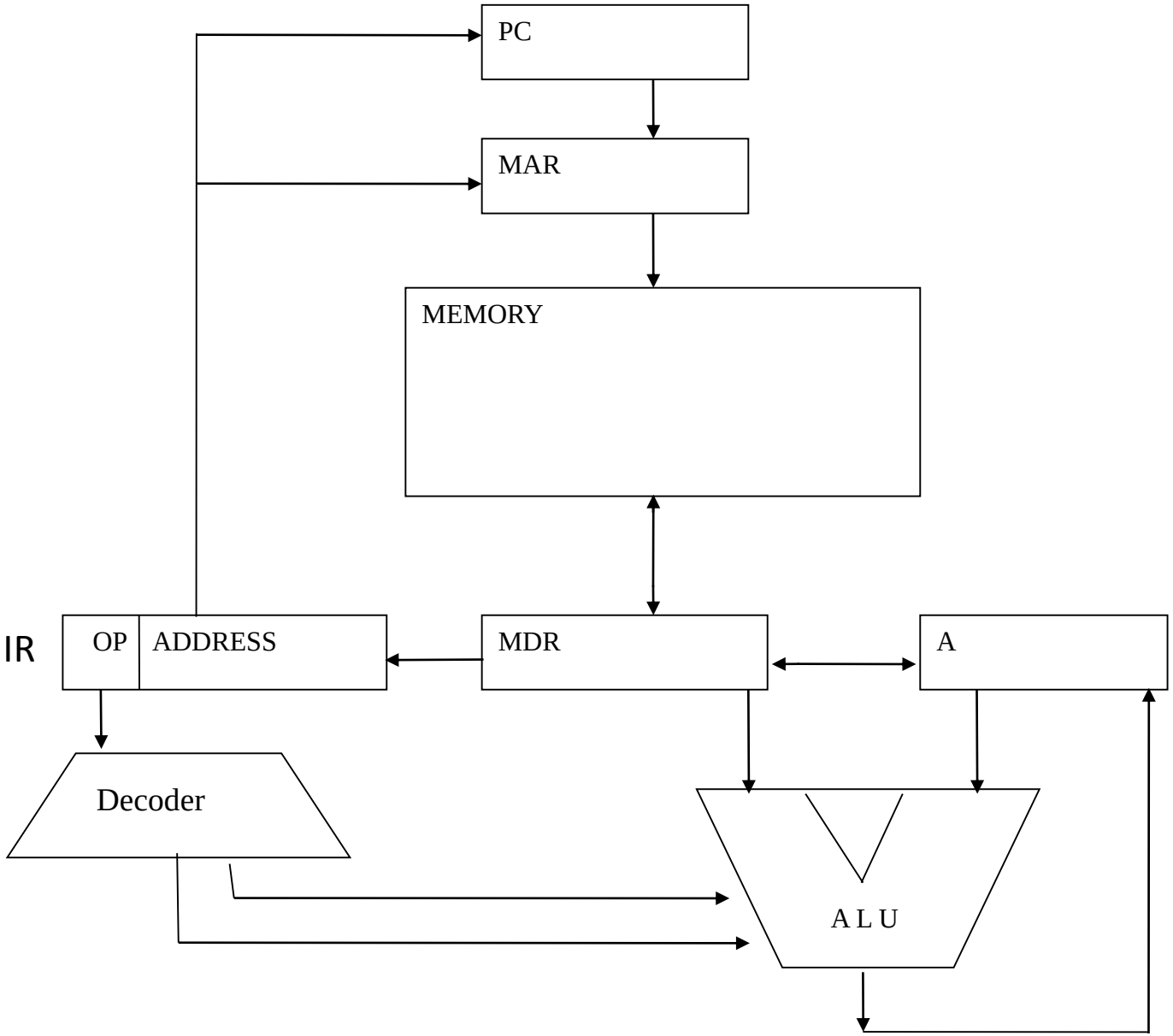
The Tiny Computer, Part I

- The Von Neumann Machine
- Definitions of the VN Components
- The Instruction Cycle
- Data Movement

The Von Neumann Machine

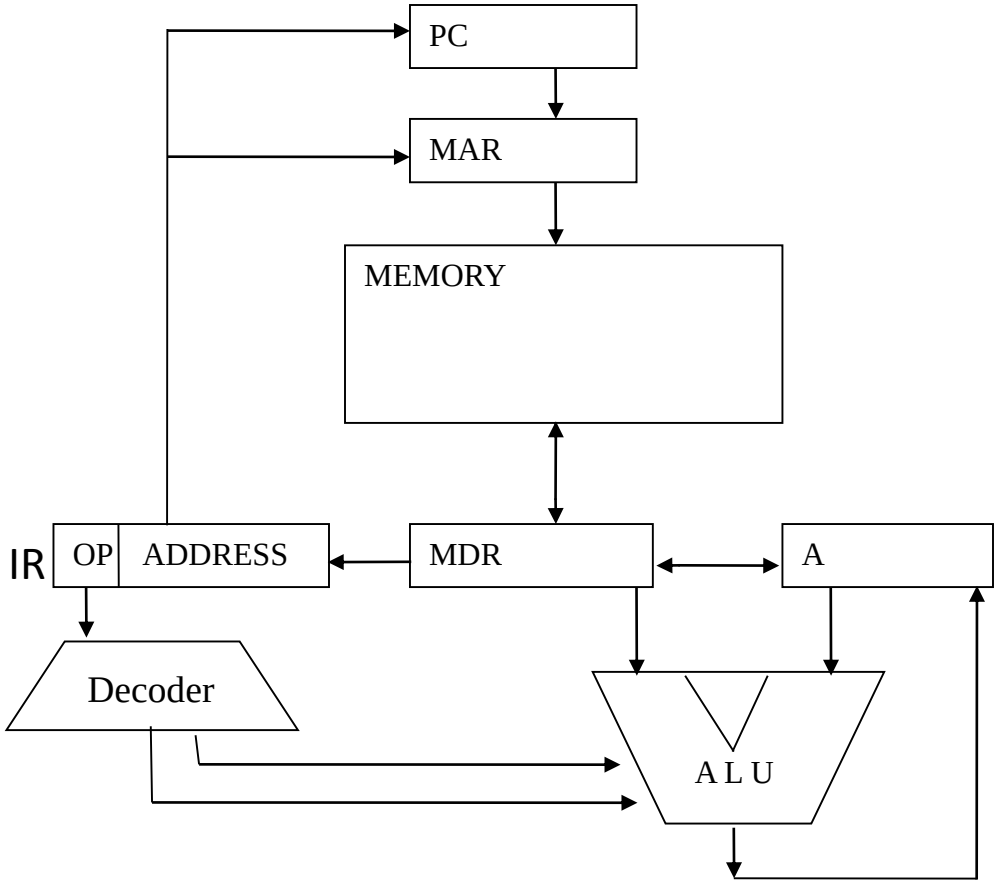
John von Neumann (1903-1957) was a polymath, highly influential in several fields. Among his contributions to computer science was a codification of the concept of stored-program computing.

It is probably technically inaccurate to call this concept the Von Neumann Architecture *– he based it on the work of Turing, Eckert and Mauchly – but his influence is such that the name has stuck.*



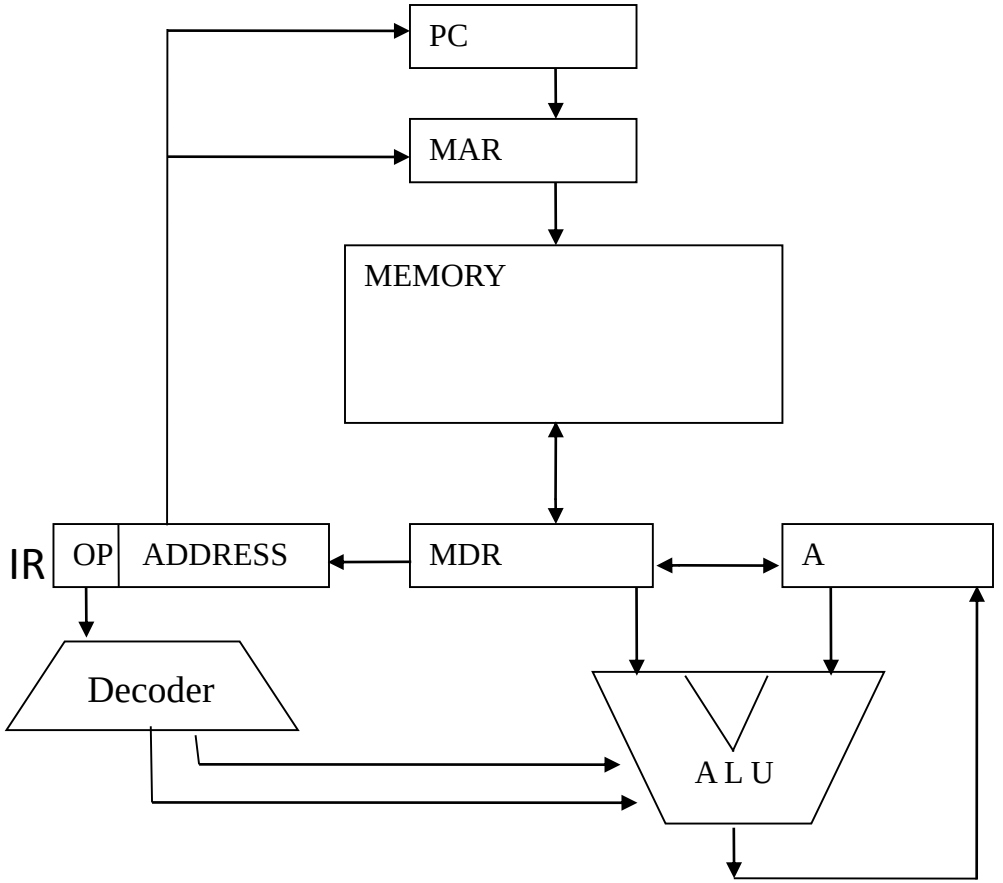
Definitions, Part 1: Components

- **Main Storage (MEMORY):** Used to store programs and data; in modern systems this is invariably implemented as RAM
- **Arithmetic Logic Unit (ALU):** Performs the actual execution of instructions
- **Decoder:** Examines the IR (next slide), decides which instruction the processor will execute and signals the ALU to execute it



Definitions, Part 2: Registers

- **Program Counter (PC):** Holds the address of the next instruction to be executed
- **Memory Address Register (MAR):** Used to specify the address to a specific memory location in main storage
- **Memory Data Register (MDR):** Used to store data being sent to or received from main storage
- **Instruction Register (IR):** Stores the instruction to be executed by the processor
  - Has two components that we'll discuss later
- **Accumulator (A):** Stores data to be used as input to the ALU



The Instruction Cycle

The *instruction cycle*, *machine cycle* or *fetch-execute cycle* of the Von Neumann machine has two looping steps:

- **Fetch Cycle:** Retrieve an instruction from memory
- **Execute Cycle:** Execute the retrieved instruction

When the instruction is finished executing, the next instruction is fetched and the cycle continues. This is a simple process to describe – but first, we need a way to describe it.

A Very Simple HDL

A *hardware description language* specifies what actually happens in processor hardware when instructions and other tasks are executed. Our little HDL is focused on data flow, with one main operator: the left arrow.

$$A \leftarrow B$$

Copy the contents of register or component B into register or component A

We will see how this works as we look at various pieces of the fetch-execute cycle. We'll start with fetching.

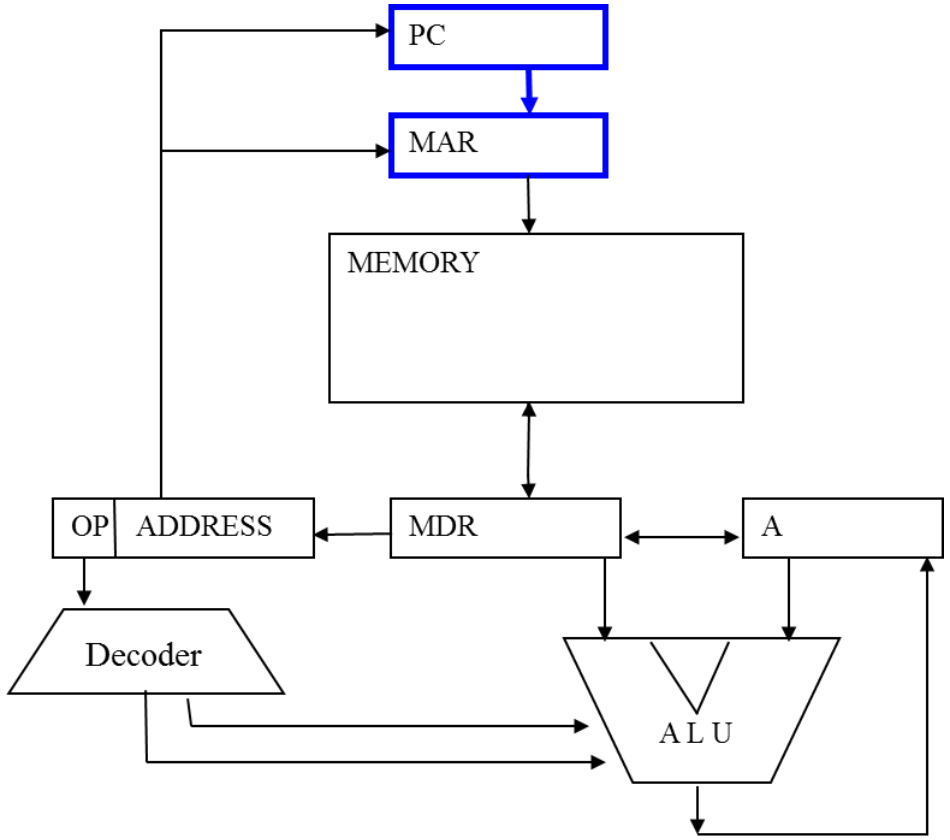
Following the Fetch Cycle, Part 1

Remember that the first part of the instruction cycle is to **fetch** the instruction. In other words, get the instruction out of memory.

The PC, by definition, contains the location of the next instruction to be executed. We need to load the MAR with that location so that we can read from it.

MAR ← PC

Copy the contents of the program counter into the memory address register.



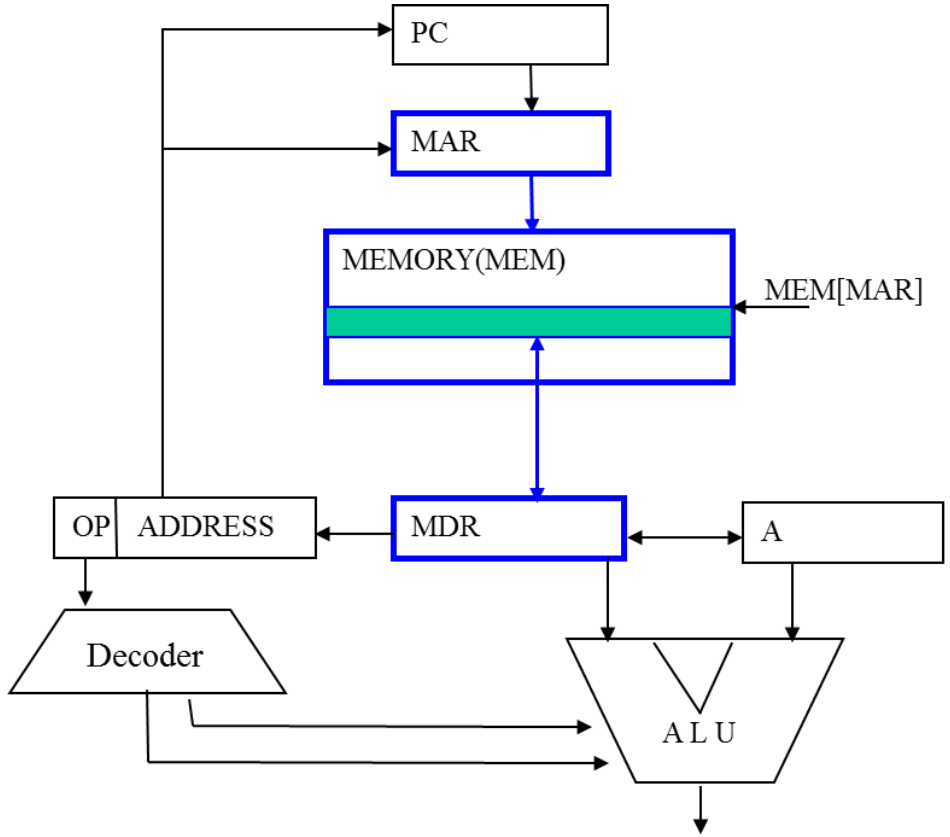
Following the Fetch Cycle, Part 2

After the last slide, we have the memory address of the next instruction in the memory access register. Now we need to read the instruction itself from memory.

We do this by reading the memory *from* the location specified by the MAR *into* the MDR.

MDR ← MEM[MAR]

Copy the contents of main memory, at the address specified by the memory access register, into the memory data register.



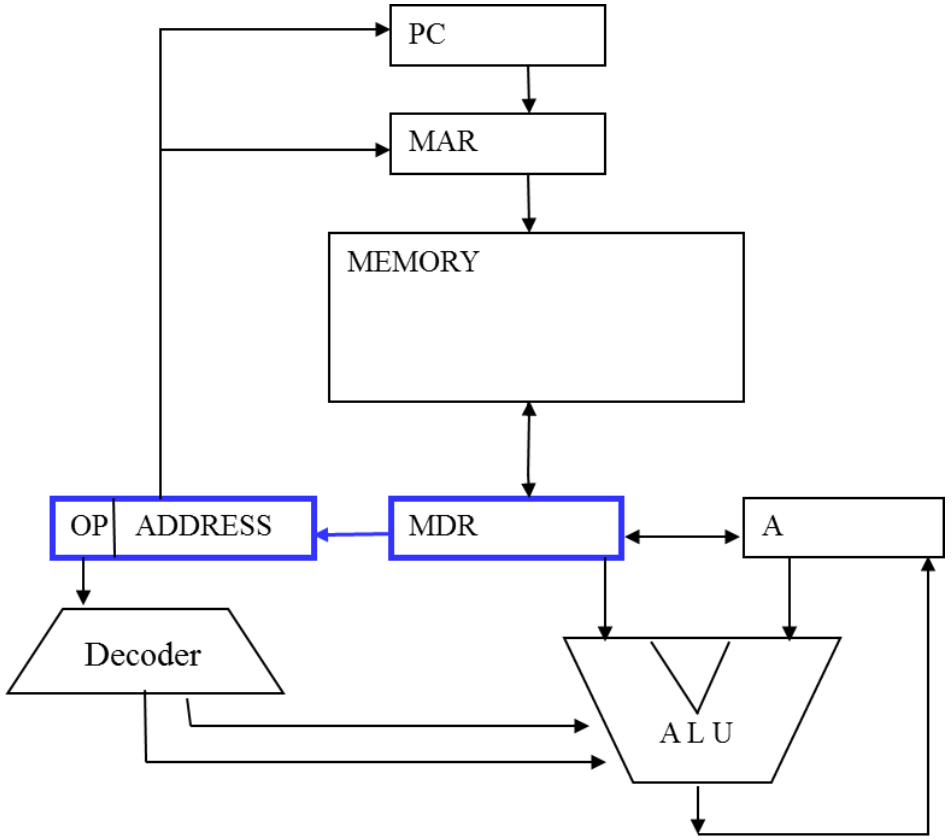
Following the Fetch Cycle, Part 3

After the last slide, we have the next instruction in the memory data register. But that's not good enough; we need it in the instruction register.

We already know how to do a register-to-register transfer from part 1, so we can just do that again.

IR ← MDR

Copy the contents of the memory data register into the instruction register.



Following the Fetch Cycle, Part 4

Recall that fetch-execute is a *cycle* – what we're doing now, we'll want to do again with the next instruction.

That next instruction most likely (though far from certainly) occupies the spot in main memory right after the one we just pulled an instruction from. So we want to increment the program counter to get it ready for the next instruction.

- Yes, now – we do this *before* executing the instruction.
- *Can you think of why?*

Any self-respecting Von Neumann architecture implementation *will* be able to increment the program counter directly, so we simply represent it as the obvious arithmetic.

PC ← PC + 1

Increment the program counter.

Digression: The Instruction Register

The instruction register has two fields: OP and ADDRESS.

- We'll call them IR.OP and IR.ADDRESS to keep it clear that they're part of the IR

Right now, we're concerned with the OP field.

- OP stands for *operation*
- It is the portion of the instruction that specifies *which instruction* is to be executed

This means the ADDRESS field is a parameter; we'll discuss it and its implications later.

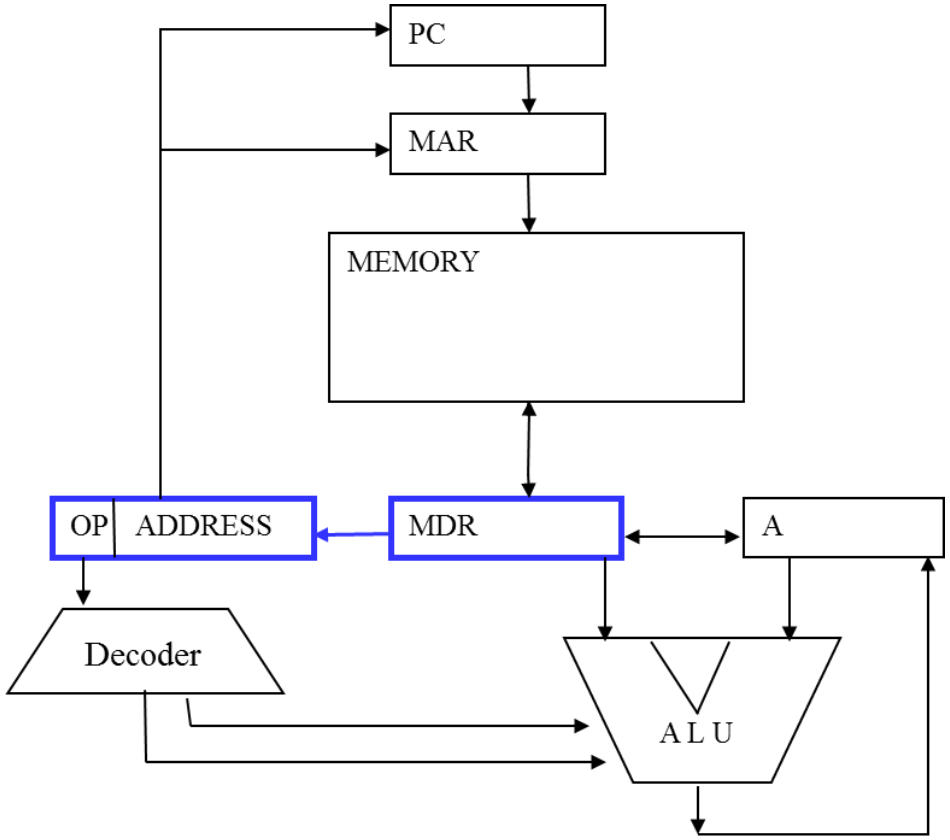
Following the Fetch Cycle, Part 5

After last slide (well, okay, three slides ago), we have the instruction in the instruction register. We now want to load that instruction into the decoder, which will actually fire off the ALU.

Recall that IR.OP is the *operation* that specifies what instruction is to be executed; that's specifically what we want to feed to the decoder.

DECODER ← IR.OP

Copy the contents of the operation field of the instruction register into the decoder.



The Fetch Cycle: Bringing It Together

MAR ← PC	*Copy the contents of the program counter into the memory address register.*
MDR ← MEM[MAR]	*Copy the contents of main memory, at the address specified by the memory access register, into the memory data register.*
IR ← MDR	*Copy the contents of the memory data register into the instruction register.*
PC ← PC + 1	*Increment the program counter.*
DECODER ← IR.OP	*Copy the contents of the operation field of the instruction register into the decoder.*

The Execute Cycle: Operations

The second half of the instruction cycle – the *execute cycle* – is if anything more straightforward than the first.

Instructions are fired off by the decoder after it's loaded from the operation field of the instruction register. Each instruction has a code, and right now, we'll look at five:

- **00: The fetch cycle, as we've just described it**
- **01 LOAD: Load from a memory location into the accumulator**
- **02 ADD: Add a memory location to the value in the accumulator**
- **03 STORE: Store from the accumulator into a memory location**
- **07 HALT: End the program**

The Execute Cycle: 01 LOAD

Load from a memory location into the accumulator. Now we see what the ADDRESS subfield of the instruction register is used for. We load the MAR, transfer from memory into the MDR, transfer from the MDR into the accumulator, and go back to the fetch cycle.

MAR ← IR.ADDRESS	*Copy the contents of the address field of the instruction register into the memory access register.*
MDR ← MEM[MAR]	*Copy the contents of main memory, at the address specified by the memory access register, into the memory data register.*
A ← MDR	*Copy the contents of the memory data register into the accumulator.*
DECODER ← 00	*Execute the fetch cycle.*

The Execute Cycle: 02 ADD

Just like loading, except when we finally get to the accumulator we add to it rather than simply load into it.

MAR ← IR.ADDRESS	*Copy the contents of the address field of the instruction register into the memory access register.*
MDR ← MEM[MAR]	*Copy the contents of main memory, at the address specified by the memory access register, into the memory data register.*
A ← A + MDR	*Add the contents of the memory data register to the accumulator.*
DECODER ← 00	*Execute the fetch cycle.*

The Execute Cycle: 03 STORE

The reverse of LOAD. We load the MAR, transfer from the accumulator into the, transfer from the MDR into memory, and go back to the fetch cycle.

MAR ← IR.ADDRESS	*Copy the contents of the address field of the instruction register into the memory access register.*
MDR ← A	*Copy the contents of the accumulator into the memory data register.*
MEM[MAR] ← MDR	*Copy the contents of the memory data register into main memory, at the address specified by the memory access register.*
DECODER ← 00	*Execute the fetch cycle.*

The Execute Cycle: 07 HALT

This simply stops the program, which is technically a new extension to our HDL: an operation that doesn't require a transfer.

STOP *End the program normally.*

Next Time: Instruction Sets and Instruction Formats