High-Performance Architecture Lectures

1. Basic Computer Organization – *What is a processor and how it works?*
   - Design of PDcLX-1 processor
2. Program Execution – *How does a Code run on a Processor?*
   - Programming PDcLX-1 processor
3. Pipelined Processor – *Increase Performance of our Processor*
   - How much speed-up with pipelined processor? What it is the cost of it?
4. Scalar Processor – *Increase Performance of our Processor*
   - PDcLX-2 and why ISA is important
5. On the way to Supercomputers – *Caches, Multicore Processor, Networks*
   - Beskow Supercomputer

Basic Computer Organization

Stefano Markidis and Erwin Laure

KTH Royal Institute of Technology

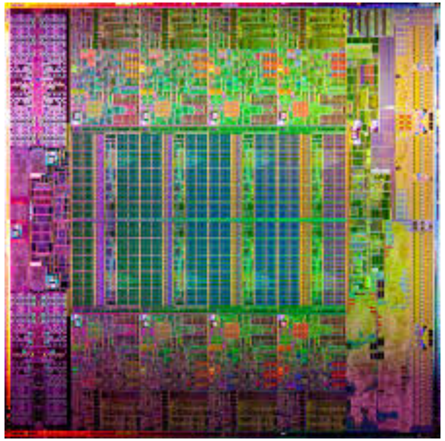
High-Performance Architecture Lectures

1. **Basic Computer Organization – *What is a processor and how it works?***
   - **Design of PDcLX-1 processor**
2. Program Execution – *How does a Code run on a Processor?*
   - Programming PDcLX-1 processor
3. Pipelined Processor – *Increase Performance of our Processor*
   - How much speed-up with pipelined processor? What it is the cost of it?
4. Scalar Processor – *Increase Performance of our Processor*
   - PDcLX-2 and why ISA is important
5. On the way to Supercomputers – *Caches, Multicore Processor, Networks*
   - Beskow Supercomputer

Microprocessor

At the heart of the modern computer is the *microprocessor* also called *Central Processing Unit (CPU)*

- a tiny, square of silicon that's etched with a microscopic network of gates and channels through which electricity flows.



The Calculator Model of Computing

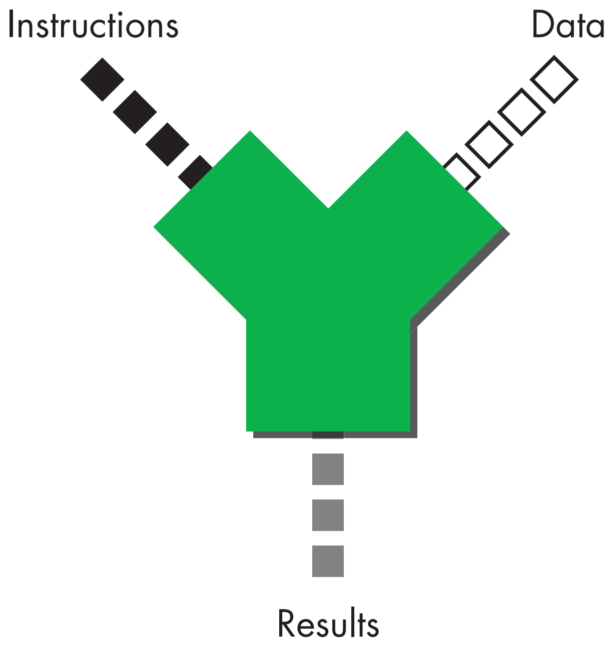
Despite computers have complex architecture, they are designed following simple principles.

A computer takes:

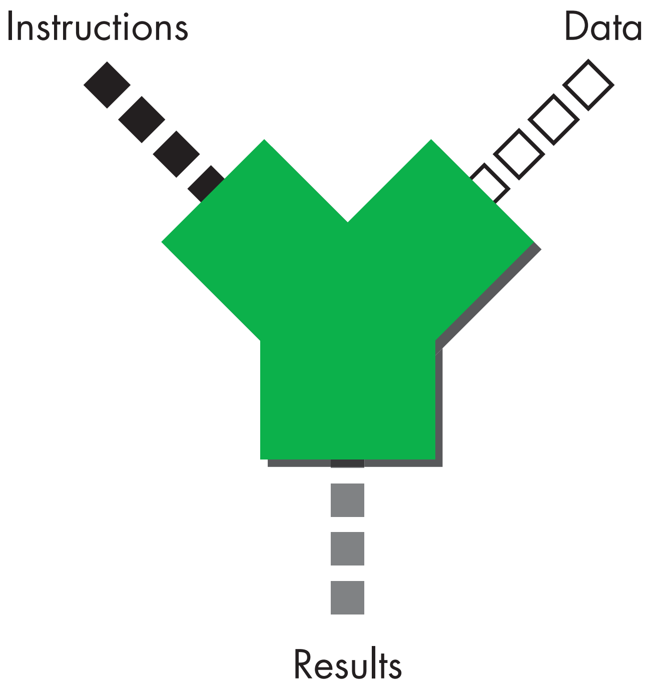
- **a stream of instructions** (code) consisting of different types of operations
- **a stream of data** as input consisting of the data on which those operations operate

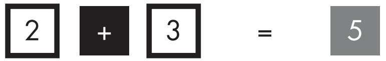
And it produces:

- **a stream of results** as output



Example: 2 + 3





Instruction stream = +

Data stream = 2, 3

Basic Computer Operations

A computer is a device that shuffles numbers around from place to place, reading, writing, erasing, and rewriting different numbers in different locations according to

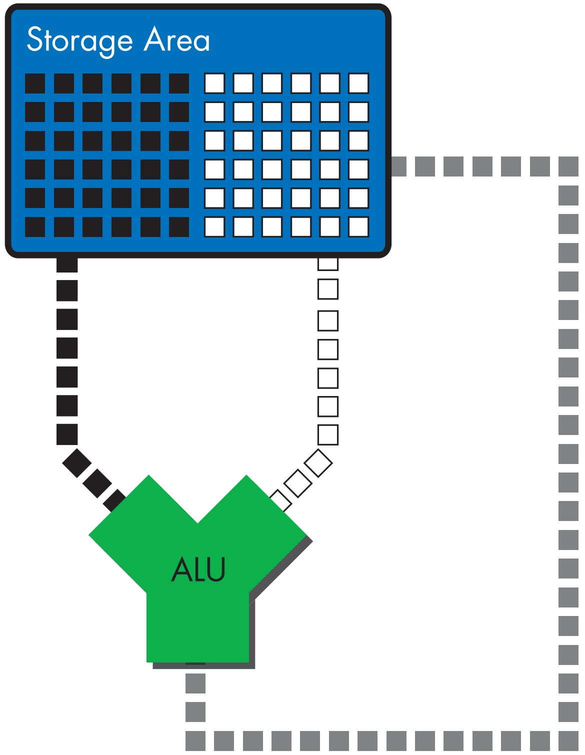
1. a set of inputs [***read***]
2. a fixed set of rules for processing those inputs [***modify***]
3. the prior history of all the inputs that the computer has seen since it was last reset [***write***]

until a predefined set of criteria are met that cause the computer to halt.

The 3 Fundamental Components of Computer

All computers consist of at least **three fundamental components** to carry out the **read-modify-write sequence**:

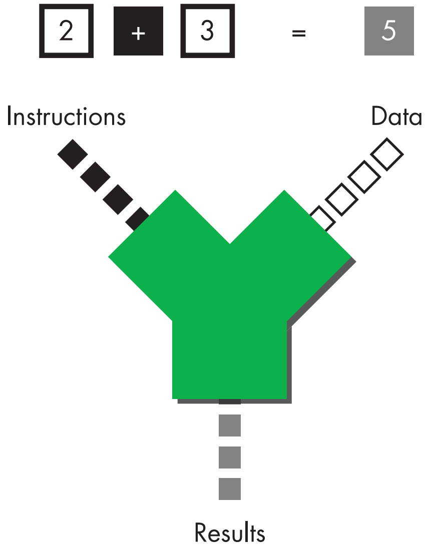
1. **Storage** To say that a computer "reads" and "writes" numbers implies that there is at least one number-holding component that it reads from and writes to
2. **Arithmetic logic unit (ALU)** and it's the part of the computer that performs arithmetic operations (addition, subtraction, and so on), on numbers from the storage area.
3. **Bus** is a network of transmission lines for shuttling numbers around inside the computer.



Instruction Stream for 2 (A) + 3 (B)

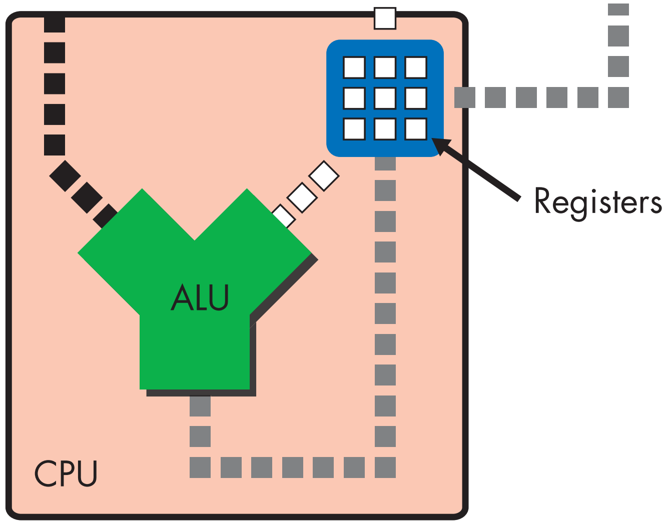
The instruction stream consists of a single instruction, an *add*, which tells the ALU to add two numbers together.

1. **Obtain the two numbers** to be added (the input operands) from **data storage**.
2. **Add** the numbers
3. **Place** the results back into **data storage**.



The Need for Registers

- We want our **data storage space to be as fast as possible**.
  - Put data storage as close as possible to the ALU
    - CPU's limited surface area constraints the size of the storage area.
  - Computers have a relatively small number of very fast data storage locations attached to the ALU, called ***registers***
    - The first *x*86 computers only had eight
    - These registers are arrayed in a a *register file*, store only **a small subset of the data that the code stream needs**



Instruction Stream for A + B = C using Registers

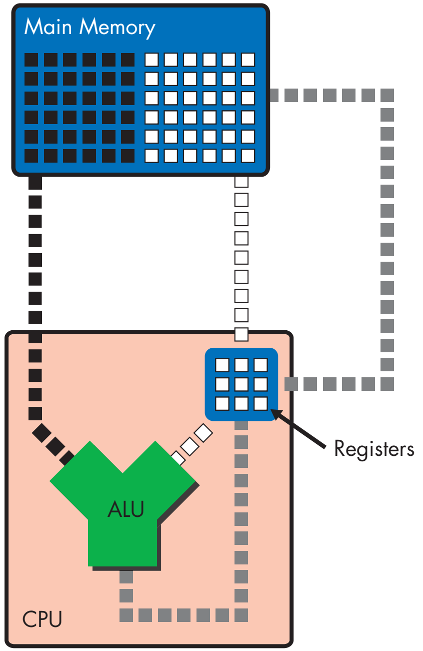
Upon receiving an instruction commanding it to perform an addition operation, the ALU in our simple computer would carry out the following three steps:

1. Read the contents of registers A and B.
2. Add the contents of A and B.
3. Write the result to register C.

Main Memory: when Registers are not Enough

We need to be able to **store very large** – register capacity is not enough.

- Main memory, which in modern computers is always some type of *random access memory (RAM)*, stores the data set on which the computer operates
  - Only a small portion of that data set is moved to the registers



What is the difference between registers and main memory?

An Example: Adding Two Numbers

To add two numbers stored in main memory, the computer must perform these steps:

1. Load the two operands from main memory into the two source registers.
2. Add the contents of the source registers and place the results in the destination register, using the ALU. To do so, the ALU must perform these steps:
   a) Read the contents of registers A and B into the ALU's input ports.
   b) Add the contents of A and B in the ALU.
   c) Write the result to register C via the ALU's output port.
3. Store the contents of the destination register in main memory

A More Precise Definition of Instruction Stream

- Instruction or code stream consists of an **ordered sequence of *instructions***.
    - **Instructions are commands** that tell the **whole computer** - not just the ALU, but multiple parts of the machine- **what actions to perform**.

Example: Adding two Numbers

If a programmer wants to **add two numbers that are located in main memory and then store the result back in main memory**, The program must consist of:

- a *load* instruction to move the two numbers from memory into the registers
- an *add* instruction to tell the ALU to add the two numbers
- a *store* instruction to tell the computer to place the result of back into memory, overwriting whatever was previously there

Different Types of Instructions

From previous example we can divide instruction in different kinds:

- **Arithmetic instructions** tell the ALU to perform an arithmetic calculation (for example, *add*, *sub*, *mul*, *div*).
- **Memory-access instructions** tell the parts of the processor that deal with main memory to move data from and to main memory (for example, *load* and *store*).
- Others (we will see them later)

Our Processor PDcLX-1

We design our own first processor!

- We call it PDcLX-1in honor of DLX architecture (https://en.wikipedia.org/wiki/DLX) and PDC ;)
  - *c* is silent so we can pronounce it as ”p-deluxe-1”
- In the next lectures, we will improve our PDcLX-1 and get to PDcLX-2.

PDcLX-1 Architecture

Our PDcLX-1 microprocessor consists

- **ALU**
- **4 registers**, named **A**, **B**, **C** and **D**

The PDcLX-1 is attached to

- a bank of Main Memory that’s laid out as a line of **256** memory cells, numbered **#0** to **#255**

Instructions Format for our PDcLX-1

Arithmetic Instruction Format

All of the PDxLX-1’s arithmetic instructions are in the following *instruction format*:

```
instruction source1, source2, destination
```

There are **four parts**:

- The *instruction* field specifies the **type of operation** being performed
- The two *source* fields tell the computer **which registers hold the two numbers** being operated on
- The *destination* field tells the computer **which register to place** the result in.

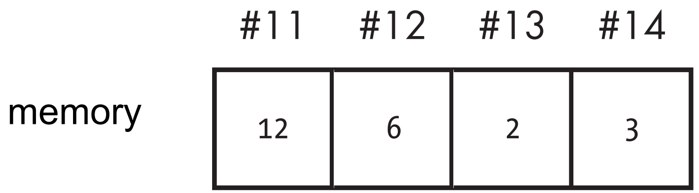
Memory Instruction Format

```
instruction source, destination
```

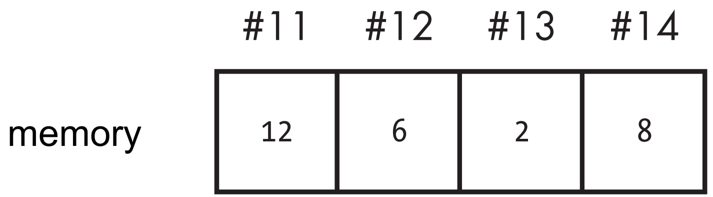
For all memory accesses, **the instruction field** specifies the type of memory operation to be performed: either a ***load*** or a ***store***

- If a *load*, the source field tells the computer which **memory address to fetch the data from**, while the destination field specifies which **register** to put it in.
- If a *store*, the source field tells the computer which **register** to take the data from, and the destination field specifies which **memory address to write** the data to.

A first Program Using PDcLX-1: A + B = C



Line	Code	Comments
1	`load #12, A`	Read the contents of memory cell #12 into register A.
2	`load #13, B`	Read the contents of memory cell #13 into register B.
3	`add A, B, C`	Add the numbers in registers A and B and store the result in C.
4	`store C, #14`	Write the result of the addition from register C into memory cell #14.



Memory Accesses: Immediate Value

All of the arithmetic instructions so far have required two source registers as input.

- it’s possible to replace one or both of the source registers with an **explicit numerical value**, called an ***immediate value***.
- To increase whatever number is in register A by 2, we don’t need to load the value 2 into a second source register
  - We can just tell the computer to add 2 to A directly

Code	**Comments**
`add A, 2, A`	Add 2 to the contents of register `A` and place the result back into `A`, overwriting whatever was there.

Immediate Values Used Before?

We have been using immediate values all along but just **not in any arithmetic instructions**

- Each load and store uses an immediate value in order to specify a memory address.
  - So the #12 in the load instruction in line 1 is an immediate value prefixed by a # sign to let the computer know that this particular immediate value is a memory address

Line	Code	Comments
1	`load #12, A`	Read the contents of memory cell `#12` into register `A`.
2	`load #13, B`	Read the contents of memory cell `#13` into register `B`.
3	`add A, B, C`	Add the numbers in registers `A` and `B` and store the result in `C`.
4	`store C, #14`	Write the result of the addition from register `C` into memory cell `#14`.

Memory Addresses in Registers? Yes!

Memory addresses **can be also stored in registers – and in memory**

- Thus, the whole-number contents of a register, like D, could be construed by the computer as representing a memory address.
  - For example, say that we've **stored the number 12 in register D**, and that we intend to use the contents of D as the address of a memory cell

Line	Code	Comments
1	`load #D, A`	Read the contents of the memory cell designated by the number stored in `D` (where `D = 12`) into register `A`.
2	`load #13, B`	Read the contents of memory cell `#13` into register `B`.
3	`add A, B, C`	Add the numbers in registers `A` and `B` and store the result in `C`.
4	`store C, #14`	Write the result of the addition from register `C` into memory cell `#14`.

Three Key-Points of this Lecture

- A computer is a device capable of reading data, modifying data and write data
- The three fundamental components of a computer are ALU, storage and bus
- We defined the architecture of PDcLX-1 and instructions format for it

High-Performance Architecture Lectures

1. Basic Computer Organization – *What is a processor and how it works?*
   - Design of PDcLX-1 processor
2. **Program Execution – *How does a Code run on a Processor?***
   - **Programming PDcLX-1 processor**
3. Pipelined Processor – *Increase Performance of our Processor*
   - How much speed-up with pipelined processor? What it is the cost of it?
4. Scalar Processor – *Increase Performance of our Processor*
   - PDcLX-2 and why ISA is important
5. On the way to Supercomputers – *Caches, Multicore Processor, Networks*
   - Beskow Supercomputer

Program Execution

Stefano Markidis and Erwin Laure

KTH Royal Institute of Technology

Goal of this Lecture

Now that we understand the basics of computer organization, it’s time to take a closer look at the nuts and bolts of **how stored programs are actually executed by our PDcLX-1**.

In this lecture we will cover core programming concepts like

- Machine language
- Programming model
- Instruction Set Architecture (ISA)
- Branch Instructions
- Fetch-execute loop

Instruction is a Binary Number!

Both memory addresses and instructions are ordinary numbers that can be stored in memory.

- A program is **one long string of numbers stored in a series of memory locations**.
- In order for the computer to run a program all of its instructions must be rendered in ***binary notation***.

Opcodes and Register Names on our PDcLX-1

We used *add*, *load*, and *store*, but they were only *mnemonics*

- Computer understands only binary!
- We map mnemonics to strings of **3-bit binary numbers**, called ***opcodes***
- Each opcode designates a different operation

We can also map the **four register names** to **2-bit binary codes**

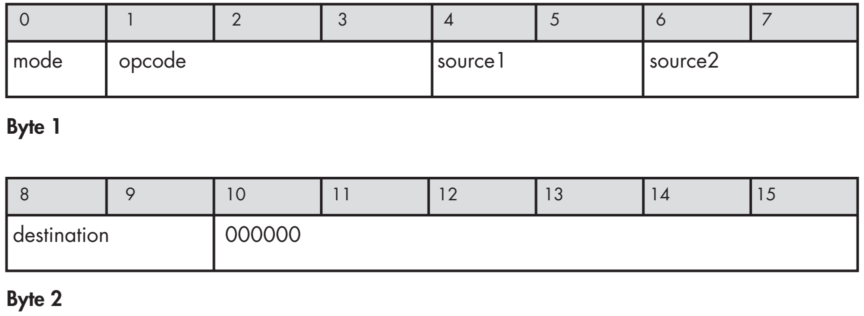
Mnemonic	Opcode
add	000
sub	001
load	010
store	011

Register	Binary Code
A	00
B	01
C	10
D	11

Machine Language Instruction

The binary values representing both the opcodes and the register codes are arranged in one of a **number of 16-bit** (or 2-byte) formats to get a complete ***machine language instruction***

Binary Encoding of Arithmetic Instructions (R)



0	1	2	3	4	5	6	7
mode	opcode			source1		source2	

Byte 1

8	9	10	11	12	13	14	15
destination		000000					

Byte 2

- **Bit 0** is the *mode bit*. If it is 0, then the instruction is a **register-type instruction**; if 1, it is of **immediate-type**. (*see previous lecture*)
- **1–3** of the instruction specify the opcode.
- **4–5** specify the instruction’s first source register
- **6–7** specify the second source register
- **8–9** specify the destination register.
- The last six bits are not needed by register-to-register arithmetic instructions, so they’re padded with 0s

Machine Code ?

Assembly Language Instruction	Machine Language Instruction
add C, D, A $\xrightarrow{?}$	
add D, B, C $\longrightarrow$	
sub A, D, C $\xrightarrow{?}$	

0	1	2	3	4	5	6	7
mode	opcode			source1		source2	

Byte 1

8	9	10	11	12	13	14	15
destination		000000					

Byte 2

Mnemonic	Opcode
add	000
sub	001
load	010
store	011

Register	Binary Code
A	00
B	01
C	10
D	11

Machine Code

Assembly Language Instruction		Machine Language Instruction
`add C, D, A`	$\longrightarrow$	00001011 00000000
`add D, B, C`	$\longrightarrow$	00001101 10000000
`sub A, D, C`	$\longrightarrow$	00010011 10000000

Consideration about Instruction Sets

Increasing the number of binary digits in the opcode increases the total number of instructions

- 3-bit opcode allows the processor to have 8, instructions in its *instruction set*
- 8 bits would allow the processor’s instruction set to contain up to 256 instructions.

Increasing the number of number of register fields increases the possible number of registers that the machine can have.

Arithmetic Instructions (I)

0	1	2	3	4	5	6	7
mode	opcode			source		destination	

Byte 1

8	9	10	11	12	13	14	15
8-bit immediate value							

Byte 2

In an immediate-type instruction, the first byte contains the opcode, the source register, and the destination register, while **the second byte contains the immediate value**.

Machine Code

Assembly Language Instruction	Machine Language Instruction
`add C, 8, A` $\rightarrow$	10001000 <u>00001000</u>
`add 5, A, C` $\rightarrow$	10000010 <u>00000101</u>
`sub 25, D, C` $\rightarrow$	10011110 <u>00011001</u>

Binary Encoding of Memory Access Instructions: Load (I)

0	1	2	3	4	5	6	7
mode	opcode			00		destination	

Byte 1

8	9	10	11	12	13	14	15
8-bit immediate source address							

Byte 2

An immediate-type load uses the immediate-type instruction format, but because the **load's source is an immediate value (a memory address)** and not a register, the source field is unneeded and must be zeroed out.

Machine Code

Assembly Language Instruction	Machine Language Instruction
`load #12, A` $\longrightarrow$	$10100000\ \underbrace{00001100}_{\text{memory address}}$

The first byte corresponds to **an immediate-type load** instruction that takes register A as its destination. The **second byte is the binary representation of the number 12, which is the source address in memory** that the data is to be loaded from.

The *Store* Instruction

0	1	2	3	4	5	6	7
mode	opcode			source		00	

Byte 1

8	9	10	11	12	13	14	15
8-bit immediate destination address							

Byte 2

The register-type binary format for a store instruction is the same as it is for a load, except that the destination field specifies a register containing a destination memory address, and the source1 field specifies the register containing the data to be stored to memory

Our Program into PDcLX-1 Machine Language!

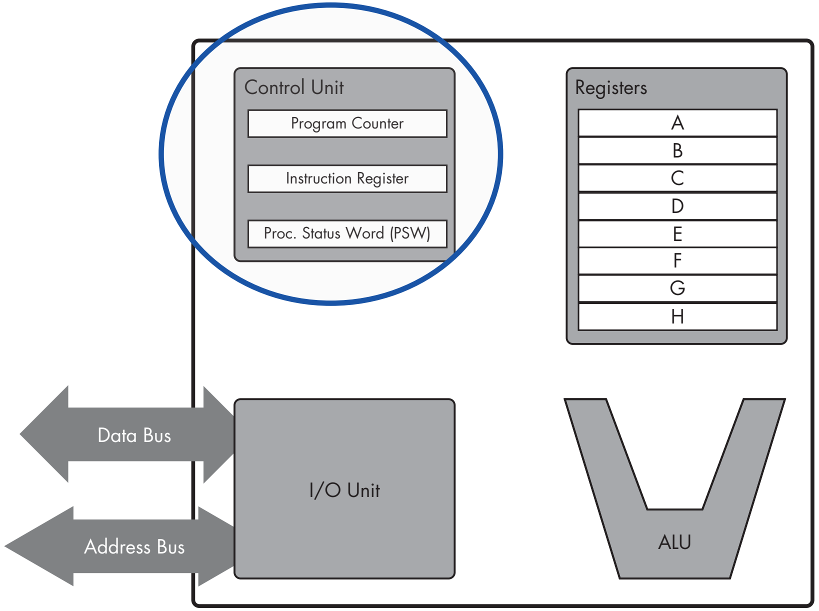
Line	Assembly Language	Machine Language
1	`load #12, A`	10100000 00001100
2	`load #13, B`	10100001 00001101
3	`add A, B, C`	00000001 10000000
4	`store C, #14`	10111000 00001110

Assembly Language

- In the first computers, programmers had to enter programs into the computer directly in machine language
- *The* ***assembly language*** **automated programming** the task of **converting** human-readable programs (mnemonics) into **machine-readable binary code**
  - Programs can be written using mnemonics, register names, and memory locations but still
    - In order to write assembly language programs for a machine, you have to understand the machine's available resources
    - a well-defined model of the machine you're trying to program

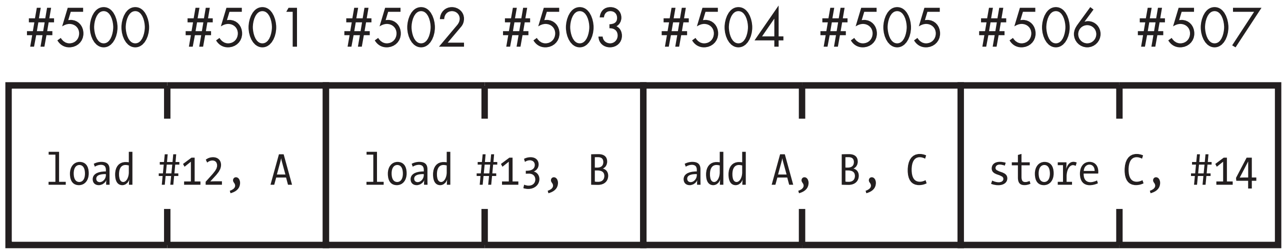
The Programming Model

- The *programming model* is the **programmer's interface to the microprocessor**.
  - It **hides** all of the processor's complex **implementation** details behind an **abstraction** exposing the processor's functionality.
- A diagram of a programming model for an eight-register machine.



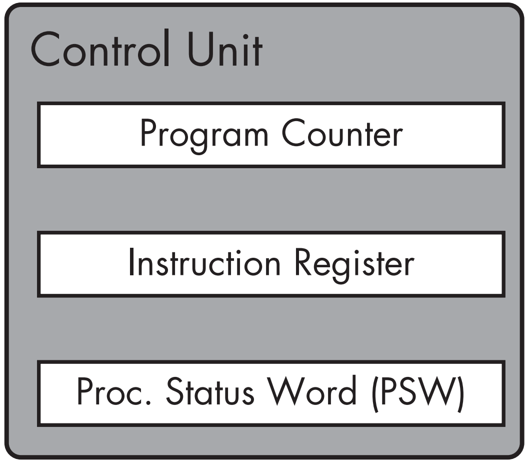
Control Unit: The Instruction Register and Program Counter

- Programs are stored in memory as ordered sequences of instructions
  - each instruction in a program lives at its own memory address
  - The instructions in our PDcLW-1 computer are two bytes long. If we assume that each memory cell holds one byte, then the **PDcLW-1 must step through memory by fetching instructions from two cells at a time**.



Control Unit and Instruction Fetch

- An ***instruction fetch*** is a special type of load that happens automatically for every instruction.
  - It always takes the address that's currently in the **program counter** register as its source and the **instruction register** as its destination.
- The control unit uses a **fetch** to load each instruction of a program from memory into the **instruction register,** where that instruction is ***decoded*** before being ***executed***



Run an Instruction

Three steps are run for each instruction:

1. ***Fetch*** the next instruction from the address stored in the **program counter**, and **load that instruction** into the **instruction register**. Increment the **program counter**.
2. ***Decode*** the instruction in the instruction register
3. ***Execute*** the instruction in the instruction register, using the following rules:
   - If the instruction is an arithmetic instruction, execute it using the ALU and register file
   - If the instruction is a memory access instruction, execute it using the memory and registers

Run a Program: Fetch-execute Cycle

These steps are referred as the *fetch-execute loop* or the *fetch-execute cycle*.

- **The fetch-execute loop is repeated for as long as the computer is powered on**.
  – The machine iterates through the entire loop, from step 1 to step 3, over and over again many millions or billions of times per second in order to run programs.

The Clock

The three steps don't take an arbitrary amount of time to complete

- They're performed according to the pulse of the clock
  - This clock pulse, which is generated by a *clock generator* module on the motherboard and is fed into the processor from the outside,
- **All three steps of the fetch-execute loop are completed in exactly one beat of the clock**.
  - To speed up the execution of programs would be to speed up its clock generator so that each step takes less time to complete.
    - Hence the race among microprocessor designers to build and market chips with **ever-higher clock speeds**.

Branch Instructions

- There are certain instructions in the instruction stream that allow the processor to **jump to a program line that is out of sequence.**
  - By inserting a *branch instruction* into line 5 of a program we could cause the processor's control unit to jump all the way down to line 20 and begin executing there (a *forward branch*)
  - We could cause it to jump back up to line 1 (a *backward branch*).

Unconditional Branch

```
jump #target
```

Unconditional branches are easy to execute, since all that the computer needs to do is

- to **have the control unit replace the address** currently **in the program counter** with **target address**

Conditional Branch and Processor Status Word Register

- The *conditional branch* instruction involves jumping to the target address **only if a certain condition is met**
  - For example, say we want to jump to a new line of the program **only if the previous arithmetic instruction's result is zero**; if the result is nonzero, we want to continue executing normally.
- Because of such conditional jumps, we need a special register or set of registers in which to store information about the results of arithmetic instructions.
  - In our PDcLX-1, this is ***Processor Status Word (PSW)*** **register**.

Conditional Branch with *jumpz*

Line	Code	Comments
16	`sub A, B, C`	Subtract the number in register `A` from the number in register `B` and store the result in `C`.
17	`jumpz #106`	Check the PSW, and if the result of the previous instruction was zero, jump to the instruction at address #106. If the result was nonzero, continue on to line 18.
18	`add A, B, C`	Add the numbers in registers `A` and `B` and store the result in `C`.

The *jumpz* instruction causes the processor to **check the PSW**

- If the bit is 1, the result of the subtraction instruction was 0 and the program counter must be loaded with the branch target address.
- If the bit is 0, the program counter is incremented to point to the next instruction in sequence (which is the add instruction in line 18).

Branch Instructions and the Fetch-Execute Loop

We can modify our three- step summary of program execution to include the possibility of a branch instruction:

- *Fetch*
- *Decode*
- *Execute* the instruction in the instruction register according to:
  - If the instruction is an **arithmetic instruction**, then execute it using the ALU and register file.
  - If the instruction is a **memory-access instruction**, then execute it using the memory hardware.
  - If the instruction is a **branch instruction**, then execute it using **the control unit and the program counter**.

How do we Start our PDcLX-1?

In its power-on default state, **the microprocessor is hard-wired to fetch that first instruction from a predetermined address in memory**.

- Pointing to the first line of a program called **BIOS** that lives in a read-only memory (ROM) module
- At the end of the BIOS program lies a jump instruction, the target of which is the location of a *bootloader* program.

Key-Points

- We defined two-byte binary encoding for our PDcLX-1 instruction (Machine Language)
- Programming Model is programmer interface abstracting processor functionalities
- Control unit (instruction register, program counter, PSW) are needed for program execution