Introduction to Computer Science – Introduction to Computer Architecture

Dr Iain Styles

October 2, 2016

1 *The Nature of Computer Programs*

The modern computer is a remarkably complex and sophisticated device, but its fundamental principles of operation have really changed remarkably little since the earliest mechanical computers, and in particular, since the IAS machine designed by John von Neumann in the 1940s. The so-called **von Neumann** architecture is at the heart of almost every modern information processing system. In order to understand why this model is so ubiquitous, it is instructive to think about the nature of computer programs and try to understand what we are really asking the computer to do. Consider the short fragment of code (written in a made-up language) in Listing 1, which computes the "factorial" function (the product of all integers up to the number entered). This function is chosen here because its program is simple enough to understand, but complex enough to have all the features that we need to illustrate how computers work.

```
a = input('Enter a number');
if a<1 exit('Invalid input');
b=1;
for(i=1;i<=a;i++) {
  b=b*i;
}
print (a,'! = ',b);
```

Listing 1: A simple program to calculate the factorial of a number

This piece of code uses constructs whose meaning is intuitively obvious to us as they look very much like natural language. Indeed, high-level languages such as C and Java are designed with this in mind. However, formally processing natural language is very difficult and computers cannot do it without significant human effort. More pertinently, the physical hardware cannot possibly "understand" such concepts.

This program, like all programs consists of two distinct "things": *data* and *instructions*. More specifically, a program is composed of a set of instructions that manipulate a set of data, and the distinction between these two types of information, and how to deal with their different processing requirements is at the root of the architecture of the computer. A very important point to bear in mind is that both instructions and data can only be stored in a binary representation, and it is not possible to distinguish them by simply examining them. The difference between instructions and data needs to be designed in to the computer.

2 *Requirements of a Computing Device*

By examining the simple program of Listing 1, we can produce a list of high level requirements that a device–a Central Processing Unit (CPU)–that could execute this program must have:

- Load the program from some external device/memory
- Receive further input from external devices

– Process instructions correctly and in the correct order
– Access and modify pieces of data in accordance with the program's instructions
– Take decisions according to the results of the computations
– Be able to repeat parts of itself
– Send the results of the the computations to some external device (screen, printer) or store them for further use

This then leads us to a list of **functional units** that implement the high-level functionality:

– Load/store instructions and data to/from the outside world
– Store instruction and data locally
– Interpret the instructions to do the necessary computations
– Send the results to external devices
– Take appropriate decisions and control what's happening
– Keep track of program execution

A block diagram of an architecture satisfying these requirements is shown in Figure 1.

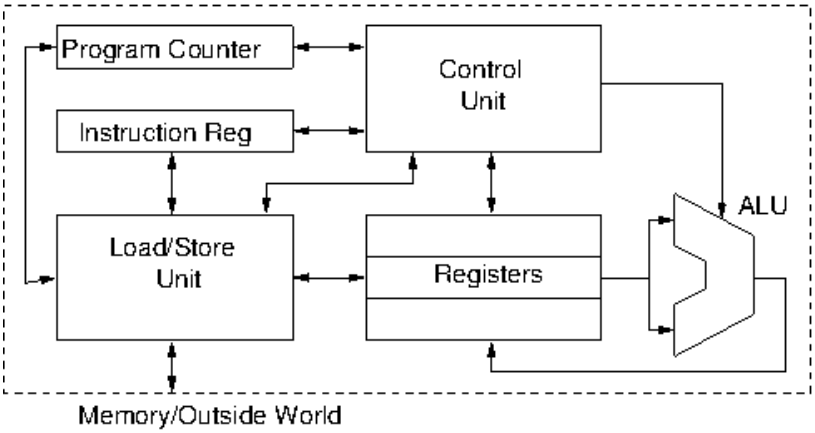

Figure 1: The basic von Neumann Architecture

2.1 *Elements of the von Neumann Architecture*

The Main Memory is physically separate from the CPU and holds all of the instructions and data that make up a program (or many programs). Within the main memory, **instructions and data are stored in distinct locations** so that they can be distinguished easily. In particular, the instructions are stored sequentially so that the flow of the program is implicit from their order. In this introductory material, we will treat the memory very simply, as a set of "boxes" each of which can hold one "item" (i.e. one piece of data, one instruction). Each box has a label which we will call an **memory address** that uniquely identifies it.

The Load/Store Unit is the interface between the CPU and the outside world, issuing and receiving requests to transfer instructions and data between the CPU and the memory.

THE REGISTERS are local, fast storage that hold data that is currently in use. Data is passed into the registers from the load/store unit. Each register holds one word of data. Note that the main registers are only used to hold data, and that instructions are dealt with separately.

THE INSTRUCTION REGISTER hold the current instruction. This is used by the control unit to configure the ALU to perform the desired computation. Only one instruction is active at any one time[1]

[1] This may not be true if special techniques for performance improvement that rely on multiple instructions being executed simultaneously have been used. We will study these later in the course.

THE ALU (Arithmetic and Logic Unit) is the computer's engine where all of the computations are performed. It takes data from the registers and updates their contents with the results of calculations.

THE PROGRAM COUNTER is a special register that contains the location (**address**)of the next instruction to be executed. It is essentially a bookmark that keeps track of which part of the program is to be run next.

In addition to these core components that reside within the CPU, there are other important components in a computer system. Because main memory is remote from the CPU, it can be slow to access and there may be an intermediate layer of memory known as a **cache** between the CPU and the main memory that is smaller but faster, and can hold portions of programs that are likely to be used again shortly. There may also be stable **long-term memories** such as disks or DVDs. These will typically require an **Input-Output (IO) controller** that handles peripheral device such as disk drives, mice and keyboards either via an extension of the memory addressing protocol, or via an **interrupt**-based protocol. The memory, the peripherals and the CPU communicate with each other via the **bus**, which carries data around and allows, for example, data to be transferred from disk into main memory. The bus consists of a set of physical wires plus a protocol (of which there are many, for example, PCI, ISA, IDE, SCSI) that is implemented by the **bus controller** that determines which subsystems can communicate (depending on priorities, previous access history, and other factors). In all cases only one piece of data can be sent on the bus at any time.

3 *The Clock Cycle*

ALL COMPUTER SYSTEMS ARE SYNCHRONOUS (with the exception of a few specialist research machine), which means that their activities are coordinated by a an external clock signal, a square-wave electric pulse (Figure 3). The speed of this pulse is frequently quoted as a measure of CPU performance (but there are many other factors and different architectures cannot be compared on the basis of their clock rate alone). Also take note that some processors

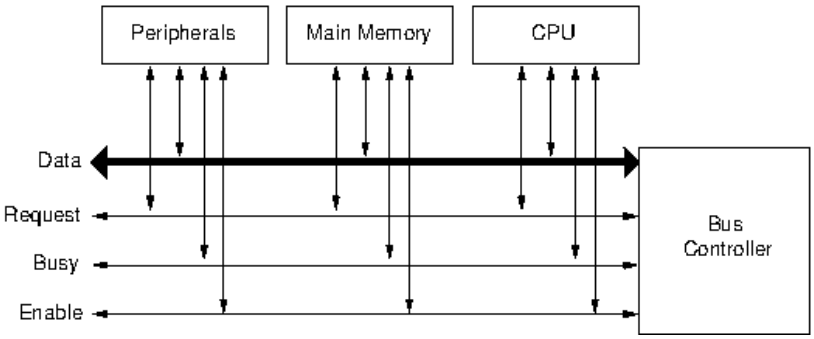

Figure 2: The Bus is the main means by which subsystems communicate with the CPU and with each other.

perform on-chip clock multiplication to generate a faster oscillation that is supplied. The time between pulses is related to the oft-stated frequency by $t = 1/f$, and so a pulse of frequency 2GHz (1 GHz $= 10^9$ Hertz) has a cycle time of 0.5ns.

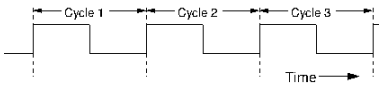

Figure 3: Waveform of a clock pulse that synchronises CPU activity.

The role of the clock cycle is to keep all parts of the computer synchronised. Variability in manufacturing means that it is not possible to know exactly how long it will take for a particular operation to complete, and so a clock cycle chosen to be slightly longer than the longest delay in the system ensures that the machine is in a well-defined state when the next set of operations start, triggered by the rising edge of the clock pulse.

The clock cycle is the global synchronisation signal that triggers new operations inside the CPU, but there are other cycles which are triggered by the clock. Of particular importance is the **instruction execution cycle**, which is the set of events that has to happen in order to for the CPU to execute an instruction.

4 *The Instruction Cycle*

The instruction execution cycle is triggered by the clock cycle, but can have several stages, triggered by successive clock pulses[2] . A complete instruction cycle usually takes several clock cycles to execute, with the exact number depending on the type of instruction and the details of a particular machine. For example, an instruction to fetch a piece of data from main memory may take several clock cycle to execute as the memory is typically much slower than the CPU, and it may take several cycles before the data is safely loaded into a register. Other instructions may complete in a single clock cycle.

[2] In some machines, some of the stages of the cycle are performed simultaneously, which speeds things up.

In real machines, the process of executing instructions is very involved. Very complex modern processors with deep **pipelines** and performance enhancements such as **out-of-order execution** require a very complex set of rules to determine what should be done next. However, most architectures follow the same basic set of stages in what is commonly called **Fetch-Decode-Execute Cycle**, a

simplified, idealised version of which is shown in Figure 4.

Fetch	1)	Inspect the program counter to find the address of the next instruction
	2)	Load the next instruction from memory into the instruction register
	3)	Update the program counter to point at the next instruction
Decode	4)	Determine the type of instruction fetched
	5)	If the instruction requires data from memory, determine its address
Execute	6)	Fetch any required data from memory into one of the CPU registers
	7)	Execute the instruction
	8)	Return to step 1 for the next instruction

Figure 4: The Fetch-Decode-Execute Cycle in Natural Language

STARTING A PROGRAM does not fit neatly into this simple model, and before the cycle is entered, we need to take a few actions to make things ready. The first thing that must be done is for the program to be loaded into main memory from disk. The instructions and data associated with the program will each occupy a block of memory (this is allocated by the **operating system**), and the memory address of the first instruction in the program is called the **entry point**. When a program is started, the entry point is loaded into the program counter. This then becomes the starting point for the cycle.

FETCH. Having loaded the entry point into the program counter (PC), we can then commence the cycle. The entry point is, of course, only the address (location) of the first instruction, and we need to *fetch* the instruction itself. At the start of the next clock cycle, the CPU issues (via the load/store unit) a request to the memory by sending the memory address and a request to read from the memory. At some later time, the instruction will be received from the memory by the load/store unit, which can then put it into the Instruction Register (IR), i.e.

```
IR ← memory(PC)
```

Note that depending on the relative speed of the clock cycle and the memory, it could take several cycles before the instruction is ready in the IR. The clock cyle and the instruction cycle are related, but they are not the same thing[3].

Once the request has been made, the value of the PC is changed to point to the next instruction. This is done by simply incrementing the program counter (PC) to point to the next *word* (box) in memory, and reinforcing the analogy of the PC as the CPU's bookmark.

```
PC ← PC + 1;
```

Some instructions can change the value of the PC in a different way. For example, **branch** and **jump** instructions are used to implement loops and conditionals, and these require the usual sequential flow of the instructions to be altered. We will see how this works later.

[3] For example, if the CPU runs at 2GHz and the memory runs at 500MHz, then it takes 4 CPU cycles for the instruction to be ready for use in the IR. A request made on cycle n will be ready at cycle $n + 4$.

DECODE. With the instruction in the IR, it can be acted upon by the CPU. First, the type of instruction is determined by the control unit. This is to determine whether any additional action needs to be taken in order to execute the instruction. The instruction could be one which moves data around (`a = b`); combines two operands (`c = a + b`); manipulates one operand (e.g. Bit shift/rotation); or does a test/comparison and changes the program flow (a branch). Other types of instruction are used to call a new procedure/function (need to store and change PC); do I/O; control loops; or do other special operations.

EXECUTE. Once the instruction has been decoded, any data needed must be fetched from memory. In most CPUs, most instructions can only access registers and there are dedicated instructions for accessing main memory. Each piece of data will have been allocated a memory address by the compiler, and this will be specified as part of the instruction. The address is extracted from the IR and passed to the load-store unit, in the same way as the instruction itself was fetched. When the data arrives at the CPU it is loaded into one of the local registers within the CPU (again, specified within the instruction). Once the data is in the registers, it can be operated upon. For example, in the simple case of `c = a + b`, the ALU will take the values of `a` and `b` from the registers that have been temporarily assigned to them, add them together, and then store the result in the register that has been temporarily assigned to `c`.

As we have already mentioned, some instructions change the flow of the program. Consider, for example the code fragment in Listing 2. The flow of the program depends on the result of a comparison, and therefore the instruction that performs the comparison must be able to change the PC. Similarly, in loops, we need to be able to change the PC back to the first instruction in the loop after each iteration is complete, and also to exit the loop when its termination condition is met. These instructions are therefore allowed to modify the PC as necessary[4].

```
if (a == b)
  //run code from address x
  PC ← x
else
  //run code from address y
  PC ← y
```

Listing 2: Code fragment showing the effect of conditional statements on the state of the PC

[4] Note that the previous value of the PC may need to be stored so that sequential execution can resume after the branch has completed.

ONCE THE RESULT OF THE OPERATION IS KNOWN, and any changes to the PC have been made, the next instruction can be executed. This follows exactly the same procedure, and we see that in principle the basic instruction cycle is quite simple (the practice is certainly not simple). We can write a simple piece of pseudo-code which describes the cycle (Figure 5).

5 *The Instruction Set*

MODERN COMPUTERS, like Babbage's Analytical Machine, are **Turing-complete** and are therefore capable of performing any computation (that can be performed). In principle, therefore, the set of

```
PC ← entryPoint                        // Set the starting point
while(programRunning){
  IR ← memory(PC)                      // Load next instruction
  PC ← PC + 1                          // Increment PC
  itype ← type(IR)                     // Find type of instruction
  if( needsData(itype) )
    dataLoc ← addrFrom(IR)             // Get data location from IR
    register(itype) ← memory(dataLoc) // Fetch data in reg specified in instruction
    }
  }
  execute(itype)                       // execute instruction
}
```

Figure 5: Pseudocode for the Instruction Execution Cycle

instructions that a computer can execute shouldn't matter, as long as they implement a Turing-complete system. In practice though, the choice of instruction set has a significant impact on a programmer's task. The **instruction set** is the set of all primitive instructions that the computer can perform, and every computer program, in any language has eventually to be decomposed into these primitives. Each type of CPU has a different instruction set and they are essentially incompatible with each other, and code written at this level is therefore strongly machine-dependent and is much harder to read, write, and debug, which is why most programming is done in high-level languages such as Java which abstract away much of the low-level detail of the machine's construction. However, the widespread use of the von Neumann architecture lends different instruction sets a certain similarity in their general form. Here, we will consider briefly the instruction set of a heavily studied processor that is popular in embedded systems–MIPS.

6 *The MIPS Architecture*

The MIPS processor is an archetypal example of a von Neumann-based RISC architecture, and follows our simple description of the von Neumann architecture quite closely (there is an instruction register, program counter, ALU etc). The basic instruction set has around 60 instructions, and MIPS is a **32-bit** architecture, and contains **31 general purpose registers**, denoted as `$1` ↦ `$31`, plus `$0` which always contains the value zero. Most MIPS instructions can only interact with the registers, and there are special instructions for transferring data to/from memory.

7 *Types of MIPS Instruction*

The instructions of the MIPS processor can be divided into three broad categories:

Memory instructions, which transfer data between registers and the memory,

Arithmetic Instructions are used for addition, subtraction etc,

and act on data held in the registers.

Control Instructions that are used to change the flow of a program,

We will focus our attention on a few of the more common instructions and how they relate to the high-level code you write. as an example, let's consider perhaps the simplest piece of code of which one could conceive:

```
a = b + c;
```

In the MIPS instruction set, there is an instruction called `add`, which takes three **operands**, which are analogous to arguments of functions in higher-level languages. The operands of `add` are: the **destination** of the result, and the **sources** of the "inputs". When programming at this very low level, one has to be absolutely specific about what is meant. The `add` instruction does not understand the meaning of variable names `a`, `b`, `c` – MIPS instructions require the **location** of the data to be specified, and for the `add` instruction, the data has to be in the registers. If we assume that `a` is in `$r1`, `b` is in `$r2` and `c` is in `$r3`, we would write this as

```
add $1,$2,$3
```

Having written this down, we realise that we have assumed that the variables `a`, `b` and `c` are already being stored in the registers `$1`, `$2`, `$3` respectively . In general, this will not be true and we will need to know how to **load** data from the memory:

```
lw $1,&a
```

This instruction loads the word at loads the contents of memory address `&a`[5] and puts it into register $r1.There is also an analogous **store word** instruction which takes the contents of a register and puts it at a specified memory address:

```
sw $1,&a
```

[5] In this module we will use this shorthand `&a` to denote the memory address of `a`, but in the real machine of course this will be the binary string representing the actual address of the item in memory.

We therefore modify our program to load in `b` and `c` from memory, and to store `a` in memory after performing the addition:

```
lw $1,&b             // Load b into r1
lw $2,&c             // Load c into r2
add $3,$1,$2     // Add the contents of r1,r2 and place the result in r3
sw $3,&a             // Store the contents of r3 at the memory location of a
```

Let's try another example:

```
a = b + c - d;
```

For this case, we will need to make use of the `sub` instruction for subtraction, `sub $x,$y,$z` which subtracts the contents of `$z` from the contents of `$y` and puts the result in `$x`. Like the `add` instruction, execution of the `sub` instruction is performed in the **ALU**.

```
lw $1,&b          // Load b into r1
lw $2,&c          // Load c into r2
lw $3,&d          // Load d into r3
add $4,$1,$2    // Add the contents of r1,r2 and place the result in r4
sub $5,$4,$3    // Subtract the contents of r3 from the contents of r4 (b+c)
sw $5,&a          // Store the contents of r5 at the memory location of variable a
```

There are also **multiply** and **divide** operations which work in a similar way:

```
mul $1,$2,$3     // multiply r2,r3, result into r1
div $1,$2,$3      // divide r2 by r3, result into r1
```

and **logical bitwise** operations

```
and $1,$2,$3      // logical bitwise AND of r2 and r3, puts the result in r1
                          // example: 01001010 & 10101001 = 00001000
or $1,$2,$3       // logical bitwise OR of r2 and r3, puts the result in r1
```

8 *Branches and Jumps*

The instruction we have seen so far either access memory or manipulate pieces of data. In most computer programs, this is not enough: we may need to take decisions about the flow of the program, repeat a piece of code many times, or execute a piece of code that is defined somewhere else. Each of these tasks requires us to break the normal sequential program flow. MIPS provides several ways of doing this, known collectively as **branches** and **jumps**.

BRANCHES occur when the flow of the program changes following a comparison between two things, as might occur in the program shown in Listing 3

```
if (i==j) {
  x = a + b;
}
else {
  x = c + d;
}
```

Listing 3: A simple program that has a non-sequential flow

Let us consider how to break this piece of code down into machine level instruction. First we must begin by loading the variables `i` and `j` into registers, which we can easily do:

```
lw $r1,&i  // load i into r1
lw $r2,&j  // load j into r2
```

If `i==j`, we execute `x=a+b`. We must load `a` and `b`, and add them together:

```
lw $3,&a  // load a into r3
lw $4,&b  // load b into r4
add $5,$3,$4 // r5 <- a+b
```

If `i!=j`, we execute `x=c+d`:

```
lw $3,&c  // load c into r3
lw $4,&d  // load d into r4
add $5,$3,$4 // r5 <- a+b
```

Note that since these two pieces of code a intended to be mutually exclusive, we can use the same registers for them. Now, we have to put these pieces of code toegther via a branch. A MIPS instruction that we could use is **branch-on-equal**:

```
beq $x,$y,&branch_dest
```

This instruction performs an equality comparison between the contents of `$x` and `$y`, and if the comparison is true, sets the program counter to address `&branch_dest`. We can use it to implement our simple conditional example. After loading `i` and `j`, we use the branch to select which piece of code to execute as in Listing 4.

```
1  lw $r1,&i
   // load i into r1
2  lw $r2,&j
   // load j into r2
3  beq $r1, $r2, L1 // r1==r2 goto L1
4  lw $r3,&c
   // load c into r3
5  lw $r4,&d
   // load d into r4
6  add $r5,$r3,$r4  // r5 <- a+b
7  L1: lw $r3,&a
   // load a into r3
8  lw $r4,&b
   // load b into r4
9  add $r5,$r3,$r4  // r5 <- a+b
```

Listing 4: Incomplete assembly

We have added a label `L1` to line 7 to indicate its memory address. If `i==j`, lines 4–6 are not executed, but lines 6–9 are. So far, this is working correctly. However, if `i!=j`, then we have a problem: lines 4–6 are executed correctly, but then there is nothing to stop lines 7–9 being exececuted. Since the two *branches* are mutually exclusive, after line 6 is executed, we need to **jump** over lines 7–9:

```
j &jump_dest // Set PC to &jump_dest
```

We need to add two lines to our work-in-progress: a jump instruction, and a jump destination. Since we still need to add a line to store `x` back into memory, so we can make this our jump destination, as shown in Listing 5.

One problem with our branch instruction is that it has completely broken the flow and readability of the code–the way the `beq` instruction works is that when the equality is true, the branch is taken, and so the order in which the two branches are written is reversed when compared to how they are written in the high-level language. A better choice would be to use `beq`'s close relation, **branch-on-not-equal**:

```
1  lw $1,&i  //r1 <- i
2  lw $2,&j  //r2 <- j
3  beq $1,$2,L1 //r1==r2 goto L1
4  lw $3,&c  //r3 <- c
5  lw $4,&d  //r4 <- d
6  add $5,$3,$4 //r5 <- c+d
7  j L2 //Jump to L2
8  L1: lw $3,&a  //r3 <- a
9  lw $4,&b  //r4 <- b
10 add $5,$3,$4 //r5 <- a+b
11 L2: sw $5,&x //store r5 as x
```

Listing 5: Assembly code for the simple if statement of Listing 3.

```
bne $rx, $ry, &branch_dest
```

Here the branch is taken when an **in**equality is true. We can therefore modify Listing 5 to use this: we exchange `beq` for `bne`, and reverse the order of the two branches:

```
1  lw $1,&i  //r1 <- i
2  lw $2,&j  //r2 <- j
3  bne $1,$2,L1 //r1!=r2 goto L1
4  lw $3,&a  //r3 <- a
5  lw $4,&b  //r4 <- b
6  add $5,$3,$4 //r5 <- a+b
7  j L2 //Jump to L2
8  L1: lw $3,&c  //r3 <- c
9  lw $4,&d  //r4 <- d
10 add $5,$3,$4 //r5 <- c+d
11 L2: sw $5,&x //store r5 as x
```

Listing 6: An alternative version of Listing 5 that maintains the ordering of the code.

In addition to these two types of branch instruction dealing with equality and inequality, there are instructions for other types of comparison involving inequalities: **branch-on-less-than-or-equal-to-zero**

```
bltz $x, &branch_instr  //branch if rx<0
```

and the close relations

```
bgtz $x, &branch_instr  // branch if rx>0
blez $x, &branch_instr  // branch if rx<=0
bgez $x, &branch_instr  // branch if rx>=0
```

These can be used in a very similar manner to the equality branches.

9 *Loops*

Branches, as we have seen, are a way of changing the flow of a program in response to some comparison. Conditional statements are not the only high-level construct where this is necessary. In loops, we have to repeat a piece of code until some condition is met as, for example, in the example given in Listing 7

```
y=0;
for (i=1; i<=x; i++){
  y = y + i;
}
```

Listing 7: A simple for loop

As we did with the simple conditional statement, let's put the assembly code together piece-by-piece. First, we need to load in all the necessary data. Actually, the only thing we need to load in is `x`, as the other variables (`i` and `y`) are both set explicitly in the code.

```
lw $1, &x        // Load x into r1
```

The next thing is to set `y=0` (we will use `$r2` to store `y`). To do this, we must make use of two things that we have not yet used. There is no mechanism for directly setting the value of a register in MIPS, but we do have a register (`$r0`) that contains the constant value 0, and an instruction for adding a constant. We therefore implement this as

```
addi $2,$0,0 // Set y=0 in r2
```

We use a similar instruction to set up the loop variable `i` in `$r3` which starts at 1:

```
addi $3,$0,1 // Set i=0 in r3
```

Now, let us break down what has to be done in the body of the loop, now that we have set up its starting conditions. Firstly, we do `y=y+i`. Then, we increment the loop variable, and test it against the termination condition. If the termination condition is met, we break the loop, if not, we go back to the start. Testing against the termination condition can be done using a branch instruction, returning to the start of the loop is a jump. We are led to the code shown in Listing 8. Notice that in order to test the loop condition (which we do by checking whether `i>x`), we have to compute `i-x` in order to compare with 0.

```
1 lw $1,&x //r1<-x
2 lw $2,&y //r2<-y
3 addi $3,$0,1 //r3<-(i=1)
4 L1:add $2,$2,$3 //y=y+i
5 addi $3,$3,1 //i++
6 sub $4,$3,$1 //r4<-i-x
7 bgtz $4,L2 //i>x break
8 j L1 //Go to start
9 L2:sw $2,&y //store y
```

Listing 8: MIPS assembly code for the simple for loop of Listing 7.

10 Summary

We have outlined the basic structure and sequences of operations that is at the heart of modern computers. It is particularly important to note that when working with software at this level, everything has to be specified by the programmer: exactly which instruction is to be used, and exactly where the data is and the result should be stored. We have seen how to implement some simple high-level language constructs in MIPS assembly language. What is particularly noticeable about what we have done so far is that it is possible to implement some quite sophisticated high-level structures using only a very few instructions. However, the resulting code listings are by no means trivial to interpret, and we begin to see why high-level languages have become so popular – a simple piece of code that we can write and intrepret without a moment's thought in a high-level lanaguage requires very careful consideration in assembly language.

11 Further Reading

[RT08, Ch. 3] covers the material from this lecture at a very general level. For detailed coverage, see [TA13, Ch. 5] and [Sta13, Ch. 13] provide good coverage of this material, but the best reference is [PH09, Ch. 3].

Bibliography

[PH09] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design: the Hardware/Software Interface*. Morgan Kaufmann, 2009.

[RT08] Carl Reynolds and Paul Tymann. *Schaum's Outline of Principles of Computer Science*. McGraw-Hill, Inc., 2008.

[Sta13] W. Stallings. *Computer Organization and Architecture*. Pearson Education Ltd, Harlow, England, ninth edition, 2013.

[TA13] A.S. Tanenbaum and T. Austin. *Structured Computer Organization*. Pearson Education Ltd, Harlow, England, sixth edition, 2013.