CS1Bh Lecture Note 23

Machines

23.1 Limitations of finite state machines

Finite state machines, while powerful, have two obvious limitations: one practical, the other more fundamental. Practically speaking, they are 'hardwired' machines. By this, we mean that they can only perform one task throughout their lifetime, such as controlling traffic lights or detecting errors on a computer network etc. Although severe, this problem can be overcome if we can easily change the wiring. For example, Field Programmable Gate Arrays (FPGAs) allow one to dynamically reconfigure the structure of an electronic circuit, allowing new FSMs to be created as frequently as we like. There is, however, a more fundamental problem with FSMs in that they cannot solve certain problems. The simplest way of describing this is that FSMs cannot count. So they cannot, for instance, take an arbitrary string of characters as input and check to see if there are an equal number of opening and closing parentheses.

There is a whole host of increasingly powerful machine models that can handle wider classes of problems, including — notably — the *Turing Machine*. For now, suffice it to say that all 'sufficiently powerful models of computing' are equivalent in power to the Turing Machine. In this lecture, we examine the von Neumann model, which is in this category.

23.2 The von Neumann machine

The von Neumann machine is the type of model that the vast majority of real general-purpose processors supports. It is based on the idea of a stored program and stored data residing in some persistent store. An instruction, pointed to by a program counter, is fetched from the program store and then executed. Depending on the type of instruction, it may read and write data values to the data store. This pattern of fetch and then execute is constantly repeated. It is known as the fetch-execute cycle, and is illustrated in Figure 23.1. The key feature of this machine, compared to any FSM, is that the behaviour of the machine can be changed by altering the instructions in the program store. We therefore say that such a machine is *programmable*. A machine-understandable form of a user program is stored in memory and is used to direct the machine's behaviour. If we wish to change the machine's behaviour, we merely change the program — not the machine.

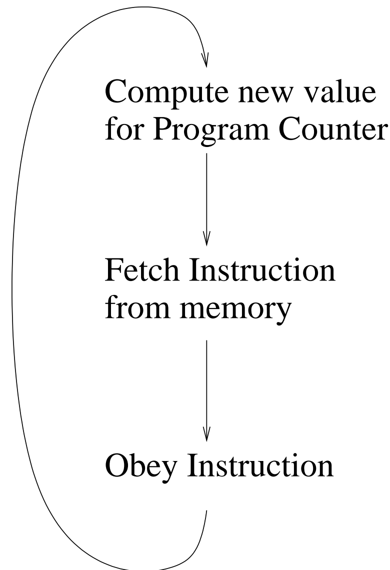

Figure 23.1: Fetch-execute cycle

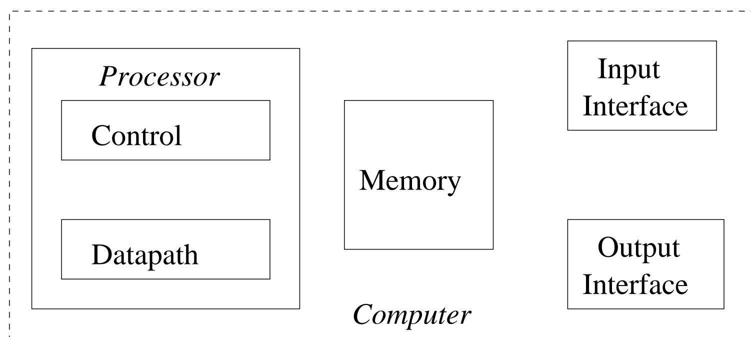

Figure 23.2: Classical model

One key aspect of the von Neumann model is that it is strictly *sequential*. That is, only one instruction is executed at a time — rather like following a recipe. This enforced sequentiality made the job of early programmers easier as they had to only consider one operation at a time. However, sequential descriptions of problems are actually neither necessary nor natural. As well as having a strictly sequential execution procedure, the von Neumann model also distinguishes between mutable data and immutable program. That is, we can change or overwrite data as often we like, but we may not change the program as we run — i.e. self-modifying code is not allowed.

The classical von Neumann model of a computer system is shown in Figure 23.2. It consists of a processor, memory and interfaces. As stated previously, the memory stores instructions and data, while the processor performs the computation, following a fetch/execute cycle. The processor can be split into two main parts: the datapath, which combines data values and performs simple operations on them, and the control unit, which manages the movement of data around the processor and the actions performed. There are also numerous interfaces to discs, keyboard, mouse, screen etc., but we will not concern ourselves at present with how these work.

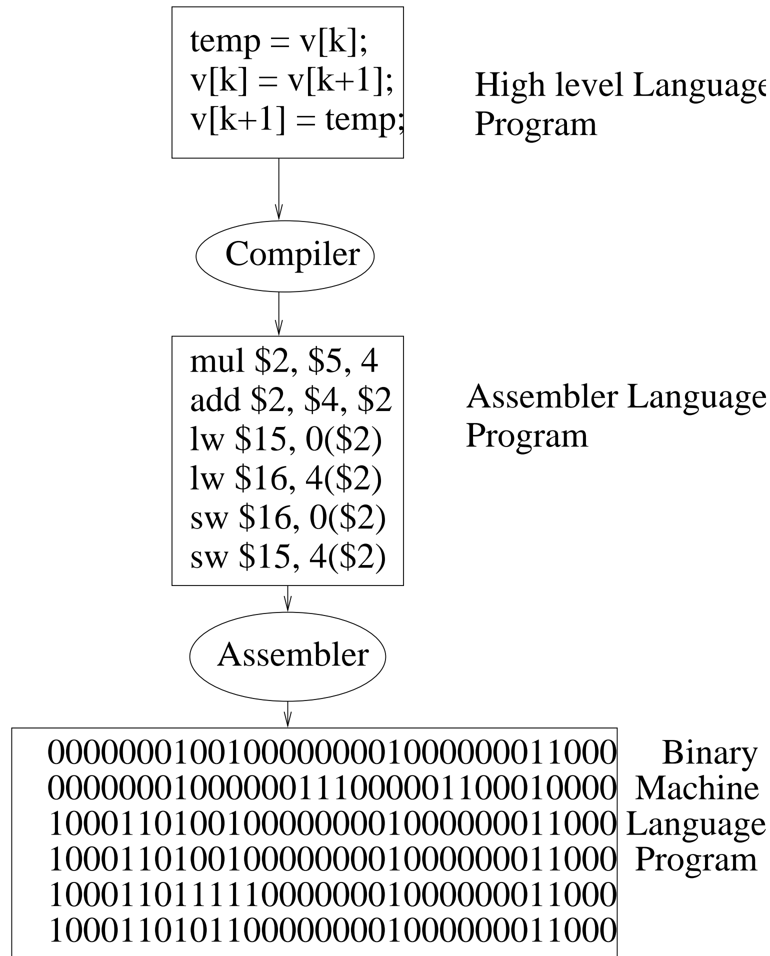

Figure 23.3: Compilation and assembly

23.3 High Level Language to Machine Code

Although we write programs in high-level languages such as Java, processors understand binary machine code — the high-level languages free programmers from the minutiae of binary instructions. There is therefore a gap between what is the programmer's view of a program and what the computer system understands as a program. This gap is bridged by compilers and assemblers, as shown in Figure 23.3. Intermediate approaches, such as interpretation and just-in-time compilation for virtual machines, have been seen in earlier lectures, notably for the Java Virtual Machine.

Each binary instruction is stored in memory before being fetched and executed by the processor. Memory consists of a series of binary digits (bits) i.e. 0s and 1s, typically organised into blocks of 32 bits known as *words*. A collection of eight bits is known as a *byte* and thus a 32-bit word contains four bytes. While today's processors usually have a word size of 32 bits, more advanced 64-bit processors have memory organised such that each word contains 64 bits. Each binary instruction is typically designed to fit exactly into a word of memory, since this is the basic unit of operation and transfer for a particular machine. This allows the entire instruction to be fetched in one go. If an instruction took up more than one word, two fetches would be required.

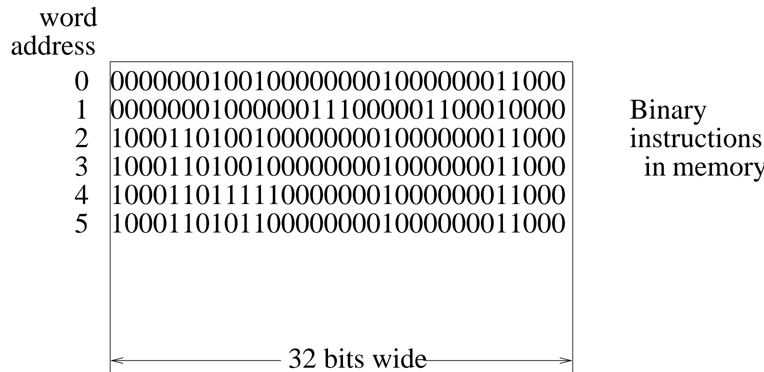

Figure 23.4: Instructions in memory

The binary instructions for the program fragment in Figure 23.3 might be laid out in memory as shown in Figure 23.4, in a state ready for execution. Initially, the first instruction fetched and executed is that at word address 0. The program counter is then incremented and the instruction at position 1 is now fetched etc. Actual implementations are a little more detailed, as we will see in the next section.

23.4 Processor and Memory

The typical structure of a processor that implements a von Neumann machine has remained relatively unaltered for 50 years. The diagram in Figure 23.5 provides a simplified view of a well-known processor: the MIPS R2000. Although now around 15 years old and slow compared to modern processors, its structure is similar to all current processors. Each processor has a well-defined set of instructions that it recognises and obeys, known as its *instruction set*. This is designed so that it is easily recognised and implemented within the processor, yet is powerful enough to execute any computable function. The essential components of a processor are:

- A **program counter** to keep track of which instruction to execute.
- An **instruction register** which holds the instruction currently being executed
- A **control unit** which decodes the current instruction and makes sure that the operation defined by that instruction gets performed.
- An **arithmetic and logic unit** for performing the basic operations defined by the instruction set of the machine, e.g. add, multiply etc.
- Several **general purpose registers**: internal storage devices (32 on the MIPS R2000, labelled \$0 to \$31) which the processor can directly manipulate. Each register holds exactly one word.
- Special **memory interface registers**. Typically, there will be a register to hold the current memory address (MAR), a register to hold data that is to be written into memory (DOUT) and a register into which data from memory is to be received (DIN).

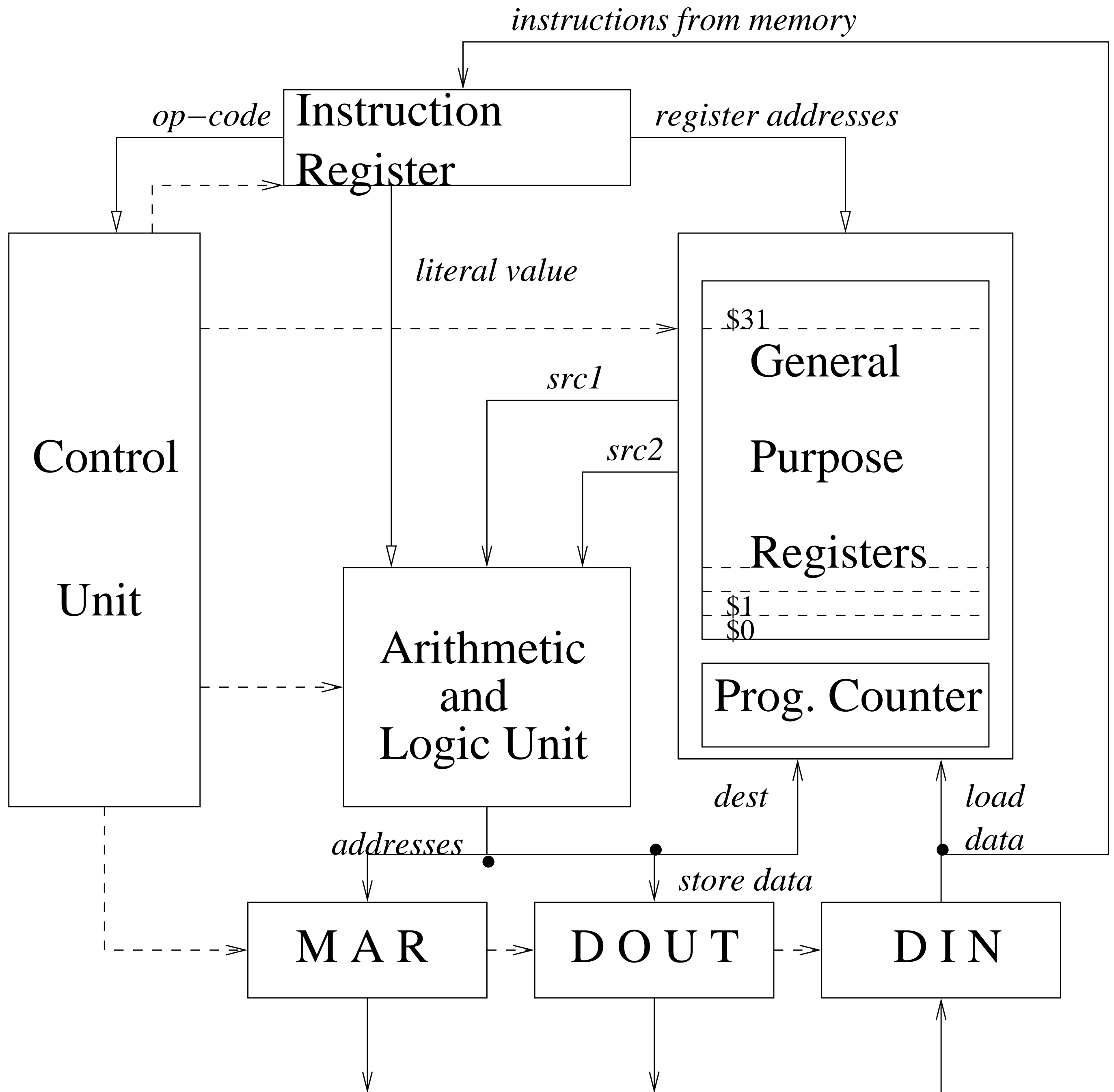

Figure 23.5: MIPS R2000 Layout

The diagram in Figure 23.5 has all of these components, and the lines with open arrowheads represent the connections which allow data to flow from one unit to another. The solid arrow-heads represent the pieces of the current instruction which are needed at various places within the processor. For example, the control unit needs to see what operation is required for that instruction, so it must get the bits from the instruction that define the operation. These bits are traditionally referred to as the *op-code*.

Typically, an operation is applied to a pair of registers (e.g. add two registers together), with the result being stored in a third register. Thus, the general purpose register file needs to know the identity of these three registers. From now on, we shall refer to these as ‘src1’, ‘src2’ and ‘dest’. The instructions contains numbers which identify these registers and those numeric identifiers are sent to the General Purpose Register File. It is often very useful for an instruction to specify a literal value. For example, if we want to add one to Register 3, we could write:

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

This instruction tells the processor to read $3 from the register file, add 1 to it and write the result back to register $3. The values in the instruction are known as the operands, and in this instruction the ordering of operands is:

<op-code> <dest>, <src1>, <src2>.

In this type of instruction, one of the operands is literally part of the instruction — the src2 operand is not a register number, but a literal value (in this case 1). To execute it, there must be a path from the instruction register (IR) to the Arithmetic and Logic Unit (ALU), as can be seen in Figure 23.5.

23.5 Instructions

In order to understand how a machine like that shown in Figure 23.5 works, we need to briefly examine the type and behaviour of its instructions. We have already met an example of an arithmetic/logical instruction — the add instruction. Other examples include subtract, multiply, shift and compare. There are two other types typically supported: data movement instructions and control-transfer instructions.

Data movement instructions load data from main memory into registers or store register values into main memory. For example, `lw $5,1500` loads the value stored at memory address 1500 into register $5, while `sw $6,100` stores the value in register $6 at memory location 100.

Control-transfer instructions are used for jumping to different parts of code and are used for iterative loops, conditional statements and returning from a function/method. For instance, if the current instruction fetched from memory is at location 5 and is `j 101`, this means that the program should jump to location 101, and thus the next instruction to be executed will be that stored at memory location 101.

We shall consider instructions, and their execution by processors, in a little more detail in the next lecture in this short series on machines.

Based on original material by Mike O’Boyle.
Gordon Brebner, April 29th, 2002.