Computer Systems and Architecture Revision Notes

James Brown

May 10, 2017

Contents

1 Introduction

These are notes I have written in preparation of the 2017 Computer Systems and Architecture exam. This year the module was run by Iain Styles (I.B.Styles@cs.bham.ac.uk). This is the module did not cover networks and they are not examinable - as such I will not be writing about them here.

2 Fundamentals of Computer Organisation

Computer programs consist of **instructions** and **data** which are identical in appearance, but they are logically distinct. Programs have a ordered set of instructions which are executed sequentially, unless it's otherwise stated. Programs also have data which is there to be manipulated by the instructions which are run. In the computers memory these will both have the same physical representation, but are not the same as each other. Because of this, when storing instructions and data they must be kept logically separate. That is to say they must be stored in different regions of memory, and not interspersed with each other for example.

We may want to describe the computers architecture at a variety of levels of abstraction:

- **Level 5**: High Level Languages. These are largely independent of the physical machine, occasionally regarded as part of the architecture.
- **Level 4**: Assembly Language. Programming in terms of the machine's basic operations.
- **Level 3**: Operating System. Common services and management functions.
- **Level 2**: Instruction Set. The basic operations that the machine can execute.
- **Level 1**: Microarchitecture. The distinct functional units that are required to implement the instruction set, and their organisation.
- **Level 0**: Digital Logic. The implementation of the functional units in terms of basic logic operations.
- **Level -1**: Physical Device. The implementation of the logic using basic electronic components such as transistors, and the physical substrate on which these are constructed.

2.1 The von Neumann Architecture and Executing Programs

2.1.1 von Neumann Architecture

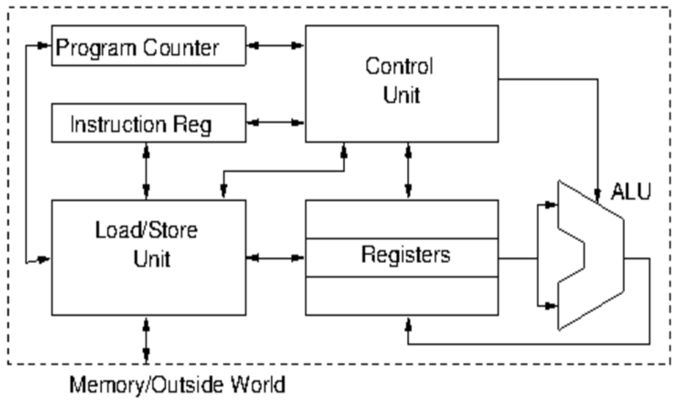


Figure 1: The von Neumann Architecture

Many modern computers are built on the (or use slightly modified) von Neumann architectures and can be considered it's heart.

We consider the main memory as being logically - but not necessarily physically - separate from the CPU. The main memory holds all of the instructions and data that make up a program(s). As stated earlier, instructions and data are stored in distinct locations within the main memory so that they are easily distinguished. Within main memory, instructions are stored sequentially so that you can determine the flow of the program implicitly from their order. Lastly, main memory is also a volatile storage method, meaning that all data is lost once the power is cut.

The **Load/Store unit** is used as the interface between the CPU and the outside world. It issues and receives requests to transfer instructions and data between the CPU and the memory via the bus.

The **registers** are small amounts of local, fast access storage the hold data that is currently in use. Data is passed to the registers by the load/store unit. Each register can hold one 'word' of data. The main registers are used purely to hold data - instructions are dealt with separately.

The **instruction register** hold the current instruction that is being executed so that it can be used by the control unit to configure the ALU. Only one instruction is active at any one time, unless the design features special techniques for performance improvements that rely on multiple instructions being executed simultaneously.

The **ALU (arithmetic and logic unit)** is the 'engine' of the computer. It performs all the computations and comparisons. It also reads data from registers and writes the results of calculations back into the registers.

The **program counter** is a special register that contains the memory location (address) of the next instruction which shall be executed - a bookmark in essence. In the normal execution of a program, the program counter is incremented after each instruction to point to the next memory location. Some instructions may change the value of the program counter in order to change the order of execution.

Additionally, we may want to add some other components just outside of the CPU which can be just as important. Due to the fact the main memory is remote from the CPU, access times can be slow. We may use an intermediate layer of memory known as **cache** that is smaller but much faster to access. This would be used to hold portions of programs that are likely to be used again shortly. We may also want to use stable, **long-term memories** such as disks or DVDs. Typically these all will require the use of an **Input-Output (IO) controller** which handles peripheral devices such as disk drives, mice and keyboards. It may do this through an extension of the memory addressing protocol or via an interrupt based protocol. Memory, peripherals and the CPU all communicate with each other via the **bus** which carries data around and allows, for example, data to be transferred from disk into main memory. This bus consists of a set of physical wires plus a protocol (there are many available, such as PCI, ISA, IDE, SCSI) that is implemented by the **bus controller**. The bus controller determines which subsystems can communicate. It should be noted that only one piece of data can be on the bus at any time.

2.1.2 The Clock Cycle

The vast, vast majority of computer systems are *synchronous* - meaning their activities are synchronised by an external clock signal in the form of a **square-wave electric pulse**. This speed is frequently quoted as a measure of CPU performance but to say it is analogous to performance would be slightly false. There are many other factors and different architectures cannot be compared on the basis of their clock rate alone. The time between two different pulses is related to the frequency of the CPU by $t = 1/f$ and is called a cycle time.

For the most part computer systems must be synchronus as variability in manufacturing means that it's not possible to know exactly how long it will take for a particular operation to complete. The clock cycle is chosen to be slightly longer than the longest delay in the system which ensure the machine is in a well defined state when the next set of operations start which is triggered by the **rising edge** of the clock pulse.

2.1.3 Executing Programs

Computer programs, in their most basic form, are just sequential series' of instructions. In von Neumann architecture, the execution of these instructions is governed by the **instruction execution cycle**. The instruction execution cycle is triggered by the clock cycle, but has several stages within it which are triggered by successive clock pulses. One complete instruction cycle usually takes several clock cycles to execute - exact numbers depend on the type of instruction and the details of the particular machine. Fetch data from memory for example may take several clock cycles to execute, and it may take several cycles before the data is safely loaded into a register. Others may complete in a single clock cycle such as the addition of the contents of two values stored in registers.

Most architectures follow the same basic set of stages in the **Fetch-Decode-Execute Cycle**. In can roughly be broken down into 8 steps in an idealised version:

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 (usually embedded in the instruction

Execute:

6. Fetch and required data from memory into one of the CPU registers
7. Execute the instruction
8. Return to step 1 for the next instruction

Starting a program doesn't fit neatly into this simple model and before we enter the cycle we also need to take a few actions to make sure that things are ready. Firstly, we need to load the program from disk into main memory. The instructions and data needed by the program will each occupy a block of memory, which is allocated by the operating system, and the memory address of the first instruction is called the **entry point**. When first started, the entry point is loaded into the program counter which then becomes the starting point of the cycle.

Fetch. Once we have a valid instruction location in the program counter (PC), we can begin the cycle. An important note is that at this point in time all we have is the memory address - not the actual instruction - we still need to actual fetch it, hence the title. At the start of the next clock cycle, the CPU issues a request via the load/store unit to the memory by sending the memory address and a request to read from the memory via the bus. Later in time, the instruction will be received from memory by the load/store unit and then stored in the instruction register (IR). 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. Once the request has been made, the value of the PC is changed to point to the next instruction - which usually just involves simply incrementing the PC. This may however be modified by some instructions such as `branch` or `jump`.

Decode. Now we have the instruction in the IR, we can begin to act upon it in the CPU. The type of instruction is determined by the control unit. This is necessary in order to determine if any further actions needs to be taken in order to execute this instruction.

Execute. After finding the type of instruction, any data needed is fetched from the memory. For a lot of CPUs, most instructions can only actually access registers and there are dedicated instructions for accessing main memory. Once the data is in the registers, it can be operated upon. As mentioned earlier, some instructions change the flow of the program and are therefore allowed to change the PC as necessary (when doing this it is often required for the previous value of the PC to be stored so that execution can resume once the branch has completed).

2.2 Harvard Architecture

In the von Neumann architecture, instructions and data are accessed via the same physical and logical pathway (the load/store unit) and there is not formal separation between data and instructions at this level. In this case, the two types of information are stored in the same physical memory but are separated by their locations within the memory. This separation is common practice in all computers as it allows for dynamic repartitioning of the memory according to the needs of the program. The problem is due to the fact that by having a shared interface, instructions and data cannot be accessed simultaneously. This is known as the **von Neumann bottleneck** and it restricts CPU performance to the rate at which is can be supplied with data.

A potential solution is to provide separate memories for small amounts of instructions and data that are likely to be used soon - separate instruction and data caches. It is also common to provide separate interfaces to instruction and data memory which is known as the **Harvard architecture**. In a pure implementation of the Harvard model, instructions and data are stored in physically separate memory but this is not flexible enough for general purpose computational devices that has a single unified memory space. The **modified Harvard architecture** has a single unified memory space (that is partitioned for instructions and data) but with separate buses for instructions and data. In most modern machines, this is the approach that is taken.

2.3 Case Studies

2.3.1 MIPS

The MIPS processor is the canonical example of a modified Harvard architecture and is very similar to the von Neumann model shown earlier. It features an instruction register, a program counter, an ALU etc, but has separate pathways for accessing instructions and for accessing data. In the diagram they are shown as physically separate, but in reality they are part of a physically unified memory.

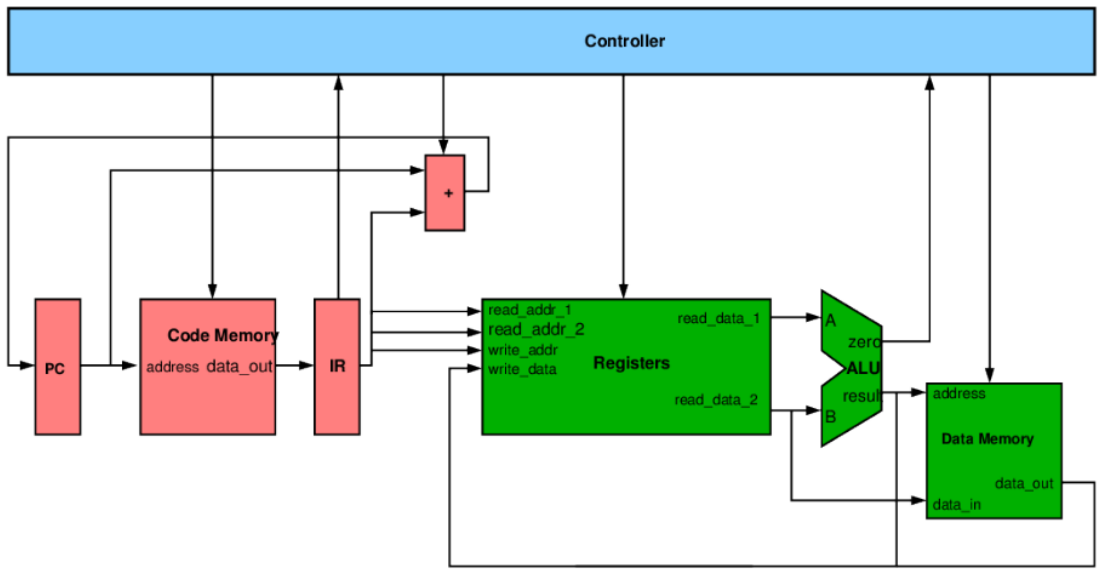


Figure 2: The MIPS processor

2.3.2 Intel x86

Modern versions of x86 are much more complex when compared to the relatively simple MIPS Harvard based architecture. This is partly down to the fact that MIPS has around 60 simple instructions whereas the latest Intel Core machines have hundreds of instructions - many of which are not simple. The benefit of these additional instructions for the x86 architecture is a significant increase in performance and it can also make it easier for the programmer/compiler writer. This all comes at the cost of a much more complex design - MIPS R4000 contains 1.2 million transistors. A quad-core i7 processor on the other hand has 731 million transistors on a die that is not much larger than that in a MIPS machine. One of the main features of modern Core architecture is that it is highly superscalar and can execute multiple instructions simultaneously. Due to this, the CPU requires several ALUs, several instruction decoders, instruction queues and multiple levels of caching. If you can see through all the complexity, you can see that the Core 2 is essentially a modified Harvard architecture.

3 Instruction Sets and Assembly Language

One of the major defining features between different computers is the choice of **instruction set**. Modern computers are *Turing-complete* - meaning they can perform any computation that can be performed - so in some sense the set of instructions doesn't matter as long as they implement a Turing-complete system. In practice, the choice of instructions can greatly affect the programmers task - especially for very low level programming.

Each type of CPU has a different instruction set which are effectively incompatible with each other. Some machines have instructions that other do not and some may have identical instructions with different binary representations. Due to this, low-level code is extremely machine dependent and written by hand only when totally necessary. Due to the widespread use of von Neumann and Harvard architectures different instruction sets tend to have a general similarity.

3.1 Types of Instruction Set

Instruction sets are frequently classified as **complex** (Complex Instruction Set Computer - **CISC**, Intel designs for example) or **reduced** (Reduced Instruction Set Computer- **RISC**, for example ARM and MIPS).

CISC computers generally have a very extensive range of instructions (this tends to be in the area of several hundered). These can range from simple instructions like addition and subtraction to more complex operations that are often common combinations of simpler instructions to provide specific support for high-level functions. CISC instruction sets have the advantage of possibly making the translation of high-level software into machine language somewhat easier. Microcoding of the complex instructions can also provide a performance benefit over their implementation in software. One drawback is the added complexity of the hardware which may make debugging and optimisation very difficult. The biggest example of CISC processors is the Intel x86 family which even includes instructions such as `AESDEC` to perform AES decryption.

RISC machines are the opposite of CISC machines - the number of instructions is minimised and each instruction is highly optimised with the ability to make use of performance-enhancing measures such as pipelining and speculatice execution. A popular RISC machine is the MIPS architecture which was very popular in the 1990s but continues to be widely used in embedded systems because of its low power and heat generation. The basic instruction set has around 60 instructions (some MIPS variants did have more), and these are **superpipelined**. Key points on the MIPS architecture:

- 32-bit architecture (instructions, memory addresses and words of data are 32 bits long)
- 32 data registers, \$0 ... \$31. \$0 is a special register and reserved for the value zero. Other registers are reserved for other special purposes by convention only.
- Most instructions can only interact with registers - there are special instructions for transferring data to/from memory. Due to this, some stages of the instruction cycle can be omitted.
- Byte-addressed, meaning that an increment of 1 in the program counter points to the next byte, not the next word. This makes the normal PC increment 4, not 1.

3.2 Types of MIPS Instructions

Instructions of the MIPS processor can be divided into eight rough categories:

Load/Store Instructions which fetch/store items from/to memory. Several variants which work on whole(32-bits) or part-words(half-words or bytes).

Arithmetic Instructions which are used to add/subtract etc. two variables being held in the registers, and also to perform comparisons.

Immediate Arithmetic Instructions which are similar to regular arithmetic instructions but used to specifically add/subtract a constant and a variable in a register.

Shift Instructions which are used to perform bit rotations - commonly used in cryptographic protocols

Multiply/Divide Instructions which perform multiplication or division on two variables being held in the registers.

Jump and Branch Instructions which are used to change the normal sequential flow of program instructions. For example, to call subroutines, take branches in the code (at conditionals) and implement loops.

Coprocessor Instructions which are used to send data and pass control to an external coprocessor which might, for example, be a graphics controller or external floating-point processor.

Special Instructions which do no fall into any of the above categories. In this course, we don't consider these at all.

To begin, we'll consider the most common instructions and how they relate to high-level code. Consider the simple code `a = a + b;`. In the MIPS instruction set, we have an instruction called `add` which takes three **operands** - analagous to arguments of functions in higher-level languages. The three operands are the **destination** of the result, and the two **sources** of the inputs. Arithmetic instructions can only access the registers so the operands must specify which registers are to be used. Therefore, we may translate our simple example into `add $8, $8, $9`. This adds the contents of the register 8 to the contents of the register 9, and stores the result into register 8. The order of the operands is important here, the destination comes first!

This code assumes the values of the variables `a` and `b` are already in the registers. We will need instructions to load data from memory into the registers and to store the result from registers into memory. An instruction to load a word of data from the memory will need operands which specify which register the data should be loaded into and where in the memory it will come from. For the moment, we will denote memory addresses using C-like notation such as `&a`. The instruction `lw $8, &a` loads the contents of a word of data at memory address `&a` and puts it in register `$8`. It follows that we also have a function `sw $8, &a` which takes the contents of a register and stores that word at the specified memory address. We can modify our program to load `a` and `b` from memory and to then store `a` in memory after performing the addition:

```
lw $8, &a;
lw $9, &b;
add $8, $8, $9;
sw $8, &a;
```

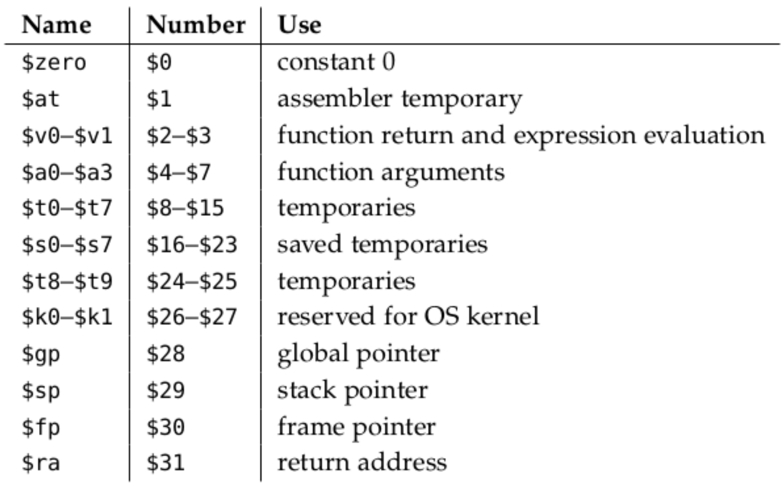
This alone does not make up a full MIPS program. We must add various assembler directives and also encapsulate this code into a main function in order to be able to run it.

3.3 MIPS Register Conventions

While all registers apart from `$0` is freely accessible by the programmer there is still a convention for their use. Their conventions are shown in Figure 3. Important registers are `$8` through `$15` which are for 'temporaries' and we can use them to store intermediate values. The assembler also supports the use of the register names rather than numbers, so those will be used from now on making to code a little more readable.

3.4 Machine Code

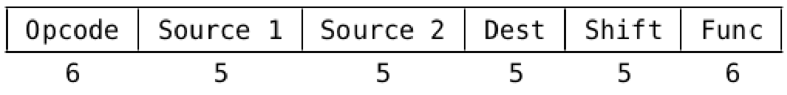
All MIPS instructions are 32 bits long, with the 32 bits divided into sections. The precise division of bits depends on the type of instruction. If we consider the simple arithmetic operations we



Name	Number	Use
`$zero`	`$0`	constant 0
`$at`	`$1`	assembler temporary
`$v0–$v1`	`$2–$3`	function return and expression evaluation
`$a0–$a3`	`$4–$7`	function arguments
`$t0–$t7`	`$8–$15`	temporaries
`$s0–$s7`	`$16–$23`	saved temporaries
`$t8–$t9`	`$24–$25`	temporaries
`$k0–$k1`	`$26–$27`	reserved for OS kernel
`$gp`	`$28`	global pointer
`$sp`	`$29`	stack pointer
`$fp`	`$30`	frame pointer
`$ra`	`$31`	return address

Figure 3: The conventions of use for all registers in the MIPS architecture

have used in our examples, the machine level representation must include information about what operation is to be performed and which registers should be used. The precise format for this type of instruction (called a **register operation** is shown below.

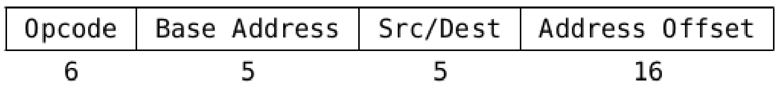


Opcode	Source 1	Source 2	Dest	Shift	Func
6	5	5	5	5	6

Figure 4: MIPS instruction format for register operations

The **opcode** field of 6 bits describes which type of instruction is being described (not the specific instruction). This is important as it determines how the rest of the instruction is going to be interpreted. The remaining bits are used to encode the two **source** registers and the **destination** register as 5 bit numbers; a **shift** field which is used by certain bit-shifting operations and denotes how far the shift should be; and finally the **func** fiedl which encodes exactly which instruction is desired. All register type instructions follow the same flow of information so having single opcode for all such instructions simplifies things significantly. We can then use the func code to configure the ALU for specific calculation. For `add` and `sub` instructions, the opcode is `000000` and the func codes are `100000` and `100010` respectively.

For **load/store** instructions, a slightly different format is required as the instructions require different information.



Opcode	Base Address	Src/Dest	Address Offset
6	5	5	16

Figure 5: MIPS instruction format for load/store operations

These instructions need to be interpreted differently to register operations and therefore have different opcodes - 35 for `lw` and 43 for `sw`. A memory address also has to be specified - which is done in two parts - a **base** and an **offset**. The 5 bits allocated to the base address do not specify an address themselves, they specify a register where the address is located. The offset field specifies an address relative to the base.

3.5 Further MIPS instructions

4 CPU Microarchitecture

The von Neumann model specifies a quite general and non-specific architecture for a computer, but absolutely none of the details. Here we are concerned with the **microarchitecture** which refers to the detailed structure and organisation of the machine. This can be divided into two broad parts:

- **The Datapath** is a collection of functional units which implement the instruction set. Each functional unit has a specific purpose: the registers are used for storing data, the program counter bookmarks the code; the instruction register stores the current instruction, and the ALU executes arithmetic and logic operations

- **The Control Logic** serves to configure the datapath in the right way so that it implements the desired instruction. It ensures that the correct data is going to the correct functional units, that the results are put in the right place and that the ALU is configured to perform the correct operation on the data. Control is the most complex part of the processor. It's somewhat simple in RISC machines due to the few operations they implement but much harder in CISC machines.

4.1 MIPS Microarchitecture

4.1.1 Instruction Fetch

The first step of the instruction cycle is to get the address of the next instruction and then to fetch that instruction. In order to do this, we need to send the contents of the program counter to memory and bring the contents of that address back to the CPU. We will need to use the **program counter**, the **main memory** and the **instruction register**. The main memory is usually not physically part of the datapath but is integral to its operation, so we will include it in our diagrams.

We can put these components together that implements the first stage quite simply. We send the contents of the program counter to the memory via the **address bus**. The returned instruction is then loaded into the instruction register via the **data bus**.

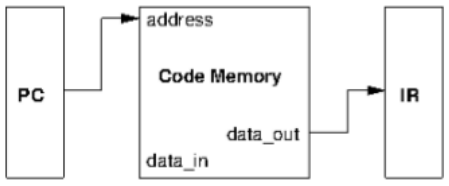


Figure 6: Datapath elements for instruction fetch

Next, we need to increment the program counter. Commonly this is just adding 4 to the PC to move to the next instruction. In the case of a branch or a jump we may need to make a further adjustment to the PC value, but the constant value of 4 is *always* added. As this is done every cycle we add a simple adder component into the datapath as shown in figure 7.

4.1.2 Instruction Execution

Whilst we should look at decoding the instruction after fetching it, it's more instructive to look at the datapath components required to actually execute the instruction set after the decoding has been done. Some components required are obvious (such as the ALU and registers), but others are less so.

Registers are required to provide a fast and local amount of memory for a small amount of data items. Any data that is used by an instruction in the MIPS instruction set must be present in the registers as they cannot access external sources (unless they are special load/store instructions). Because of this, we need to provide mechanisms for writing data into the registers and also for reading data from the registers. For some instructions we may need access up to three registers

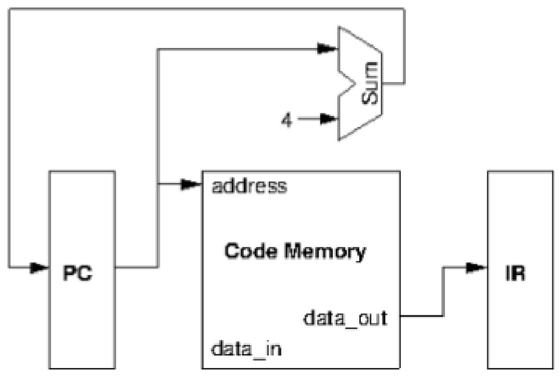


Figure 7: We have integrated an adder to increment the PC

simultaneously. As MIPS has 32 registers, we can use a 5-bit number to uniquely address a single register and these are encoded into the machine code for the instruction. The registers must therefore feature three 5-bit address ports - two of these are for reading and one is for writing - and two 32-bit outputs for retrieving data from the registers. We will also need one final 32-bit input to write data to registers. As not all instructions write to registers we also include an input which signals whether the register specified by the `write_addr` is to be written or not. All of these are shown in Figure 8.

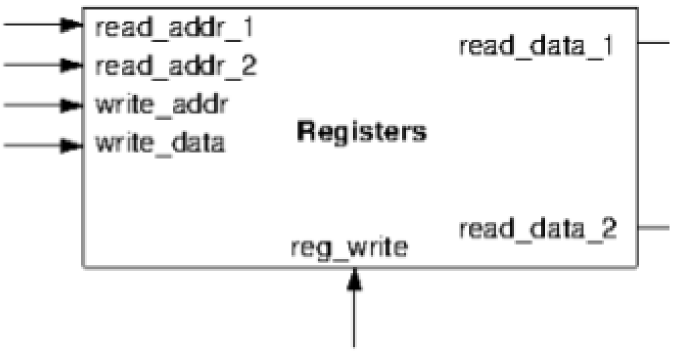


Figure 8: Registers and their input/output ports

Next, we would like to build the ALU. The ALU performs all of the processors arithmetic and logic operations and it's internal circuitry must be able to be configured so that it can perform any of the "ALU instructions". In MIPS, the ALU must also be able to compute address, for example, whena address offsets are used - e.g. `lw $0, 8($sp)`. The ALU handles the addition of the offset. The ALU also handles comparissons required by by branch and set instructions. The ALU must be able to accept up to two 32-bit words from the registers as input, perform the ALU instruction specified in the instruction register as determined by the ALU control unit and to output another 32-bit word (in the case of comparissons, just true/false using the zero output). The ALU is shown in Figure 9.

The registers and parts we have shown so far form the basis for a datapath that can execute R-type instructions (e.g. `add`). Fifteen bits are dedicated to selecting registers that are involved in ALU computations. We can pass these register addresses directly into the register address ports to select the correct registers. The remaining 17 bits are used by the control to configure the datapath to perform the instruction. This gives us the datapath shown in Figure 10 for these R-type instructions.

Load/store operations are somewhat more complex instructions. They need read and write access to both the registers and main memory, as well as to be able to use the ALU to calculate addresses. A source/destination register address is specified dependent on whether the operation is

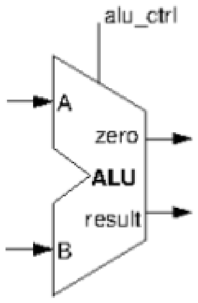


Figure 9: The ALU and it's input/output ports

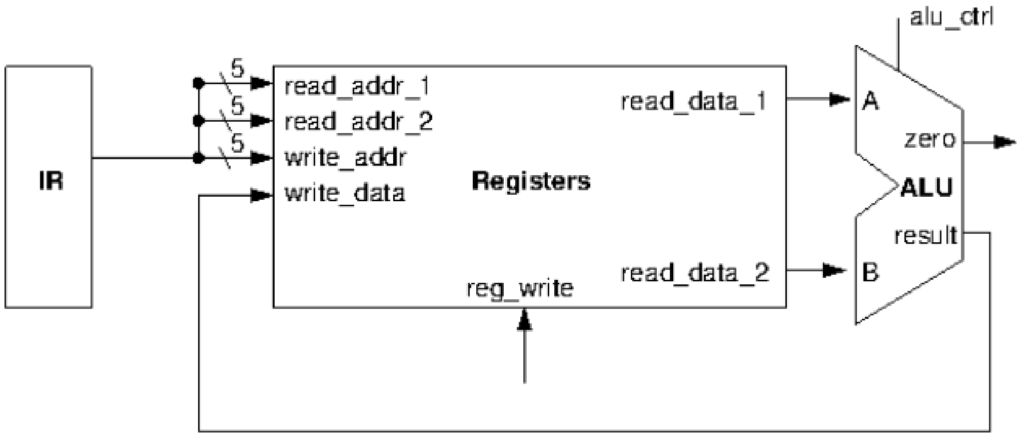


Figure 10: The datapath for R-type instructions

a load/store. The register of the **base address** is also specified and the 16-bit address **offset**. The addition of the base address and the offset is performed by the ALU in the MIPS architecture, but is not as straightforward as it might seem. The base address is 32-bits, whilst the offset is 16-bits. We need to add a new unit to 'extend' the offset so it can be added to the base. This seems trivial (just prepend some zeroes), but branches can go backwards and forwards in the code so we must allow for negative offsets using the two's complement representation. The manipulations that are required to translate the offset are performed by the **sign extension unit**. Once the address is computed by adding the offset to the base address, we must read or write the memory address that we have computed, as well as update the contents of the specified register is we are loading. A signal to the memory that controls whether is is a read or write that will be generated by the CPU control unit. We omit this for now to focus on the flow of data.

4.1.3 Branches and Jumps

Both branches and jumps achieve essentially the same result - they both have to change the contents of the program counter. The datapath mechanisms are essentially the same because of this, but the control differs depending on the specific instruction being executed as branches require a comparisson to be done. We will deal with branches first, and add jumps later.

In order to execute a branch, we have to look at the type of instructions, which is the same as I-type load/store instructions. In the `beq` instruction for example, we must compare the contents of the two registers specified in the instructions and if they are equal, we add the specified offset to the program counter. Otherwise we increment the PC by 4 as normal. The ALU is capable of performing comparrisons, and the registers are used to store the quanitites to be compared. We will also have to extend the offset to 32-bits from 16-bits in order to add it to the PC. As we need the ALU for the comparisson, we use a dedicated adder instead of the ALU for the bit extension.

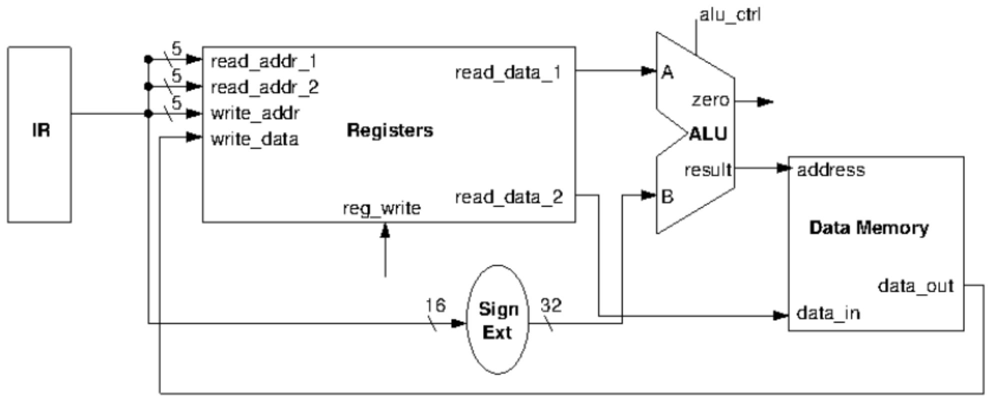


Figure 11: The datapath for load/store instructions

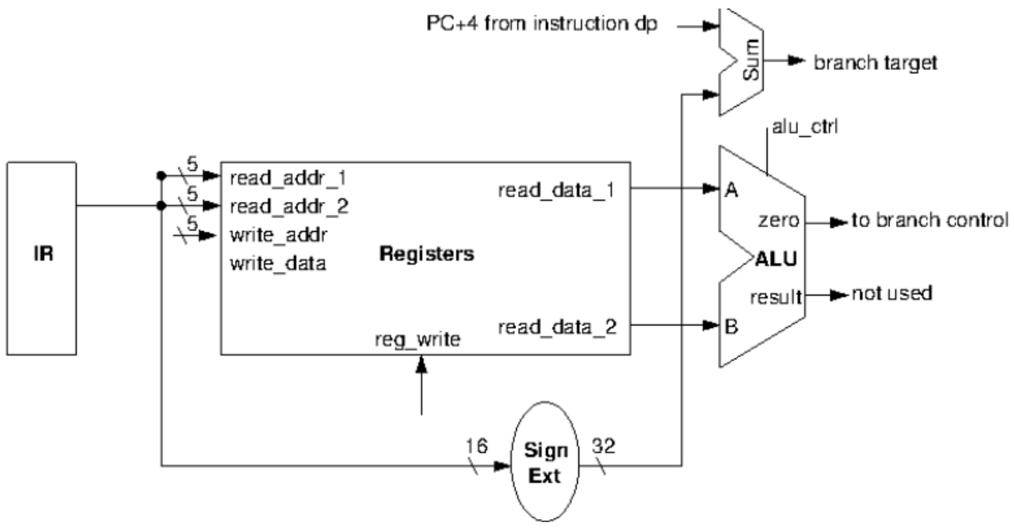


Figure 12: Datapath for branch instructions

This incurs a little overhead but reduces overall complexity. The resulting datapath is shown in Figure 12.

4.2 Integrating the Datapath

We've constructed datapath circuits for a range of the most common MIPS instructions, each of which has their own special requirements. Many of them also have a lot in common, and many of the functional units are used in more than on instruction type. Duplicating functional units is clearly undesirable and because of this we need to be able to control the flow of data within the datapath to ensure that the correct units are employed for each instruction. An example of why this is difficult is that data can be written into the registers from either the ALU or from the data memory, but only one set of wires which these pieces of data can be sent and we cannot simultaneously send two pieces of data down the same wires. The solution is to add switches to the wires, which is known as **multiplexing**.

A 2-1 multiplexer is a switch which allows one of the inputs to pass, based on a control signal C. With N control wires, we can choose 2^N inputs. For example, an 8-1 multiplexer will require 3 control wires. Multiplexing will allow us to combine and integrate different sections of the datapath that we designed for the different types of instruction. Firstly, we will integrate the ALU-based instructions which use the IR, the ALU and the registers; and the load-store instructions which use the IR, the registers, the ALU, data memory and a sign extension unit. The challenge is to integrate these two datapaths using only one ALU, and one set of registers. The two datapaths

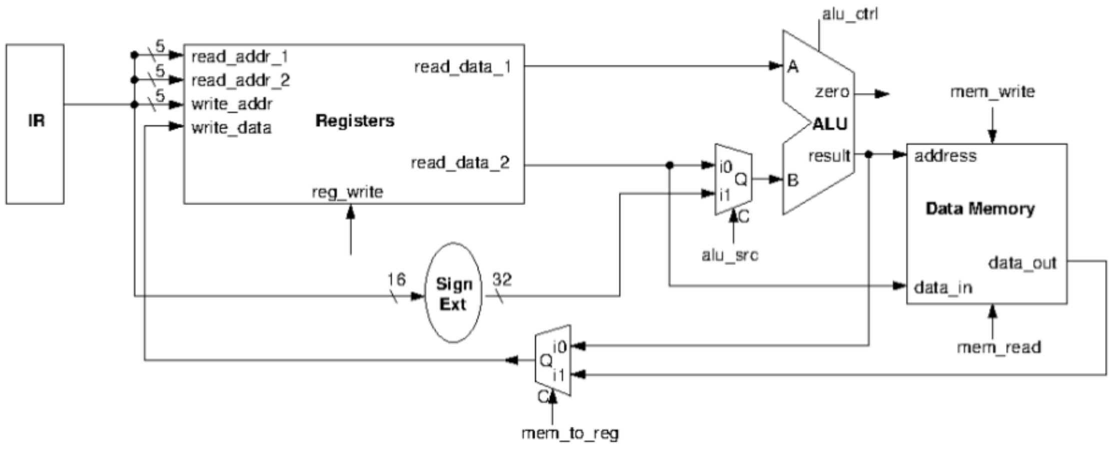


Figure 13: The integrated datapath for ALU and load/store instructions

we are trying to integrate can be seen in Figures 10 and 11.

In an ALU instruction, the ALU inputs both come from the registers, and the result of the operation is written back to the registers. In a load/store instruction, the ALU inputs are from the registers and sign-extension unit, whilst the registers need to be written by the data memory. Therefore we need to multiplex the data that is to be written into the registers; and the ALU inputs, as show in 13.

The introduced multiplexers that have been introduced are controlled by two signal, `alu_ src` controls whether ALU input B is from the registers (during an ALU instruction) or form the sign extension unit (during a load/store); `mem_ to _ reg` controls whether the ALU (during an ALU operation) or the memory (during a load) writes to the registers. The CPU control will set the state of these according to which type of instruction is being executed. Since it's only during stores taht the memory is being written, the control must also set the relevant control signals (`mem_ read`, `mem_ write`).

Now we can integrate the instruction fetch datapath, which is relatively straightforward to incorporate, with no multiplexing required. This is shown in Figure **??**.

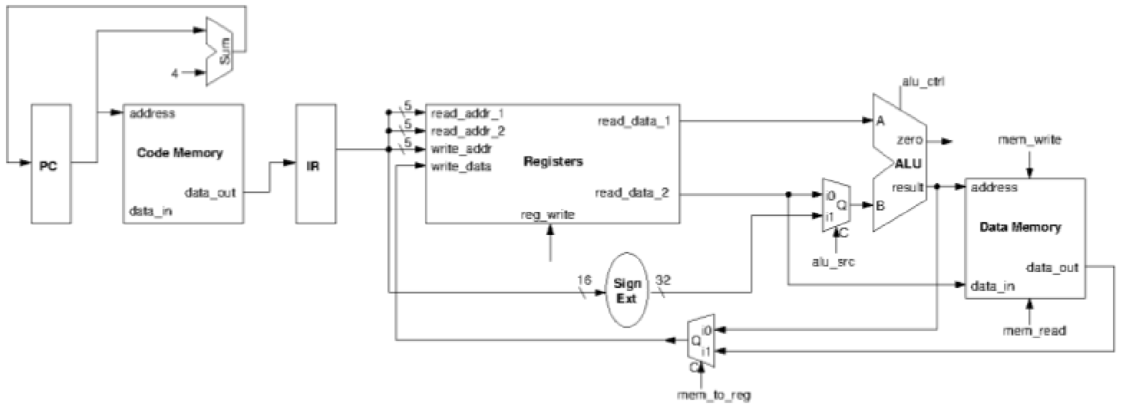


Figure 14: The integrated datapath for the ALU and load/store instructions, and instruction fetch

In order to add branches to this datapath, we need to add an adder which will add the branch offset to the PC. In normal operation the PC is incremented by 4 each cycle, whereas in branches it is updated by 4+`offset`. As we need to be able to select which value the PC will take, we insert a multiplexer to select between them as shown in 15.

We haven't yet included jumps into our datapath. This is because it is easier to do this once we have integrated things together. We will only consider a 'plain' jump rather than the more

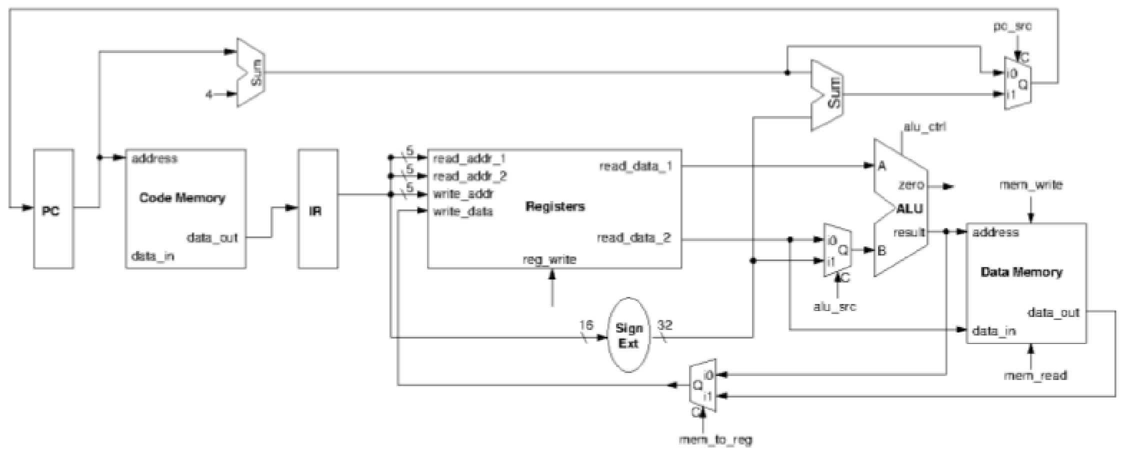


Figure 15: Integrated datapath for ALU and load/store instructions, instruction fetch and branches

sophisticated jump-register or jump-and-link instructions for now. In the jump instruction, a 26-bit memory address is encoded in the instruction. This is assumed to be a word address not a byte address, so it first must be converted into a byte address by shifting it left by two bits. Following this, it is concatenated with the four most significant bits from the PC to form a full 32-bit address. Due to this, the jump instruction can therefore span up to 256MB relative to the current position in the code. We then multiplex the jump address with the PC update value from the branch multiplexer and selected using the 'jump' signal.

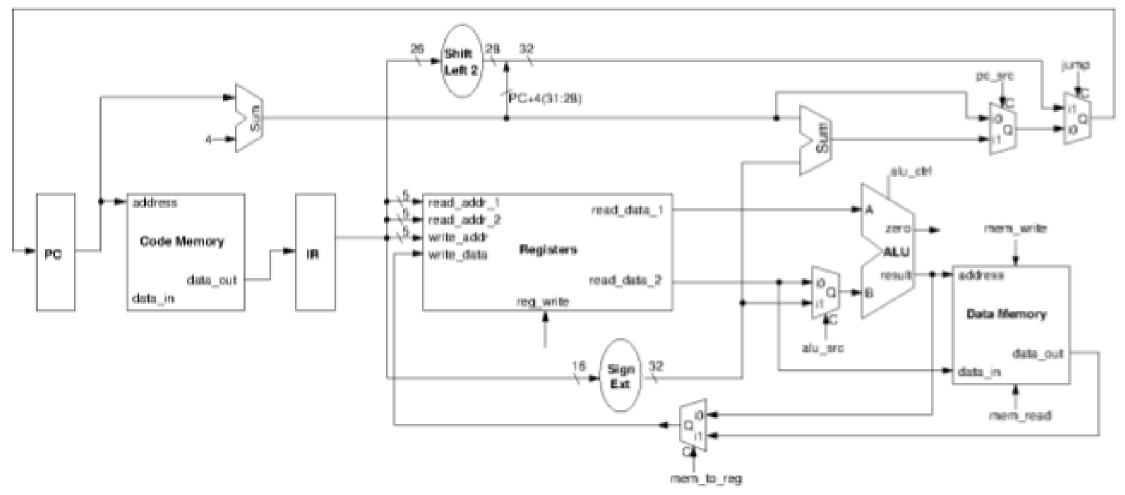


Figure 16: Integrated datapath for ALU and load/store instructions, instruction fetch, branches and jumps

Our design for a datapath can now execute a small subset of MIPS instructions. Incorporating other instructions follows a similar process (for example, jump-and-link requires the PC to be fed into the `write_ data` port of the registers, with the `write_ addr` set to 11111 (for $31)). There are a couple of modifications we need to make to this datapath in order to truly accurately represent the MIPS design. Firstly, different instruction formats must be fully accomodated. Bits 31-26 contain the opcode, but the role of the rest of the bits changes between the different instruction types. In particular, in R-type instructions the destination register address is given by bit 15-11, whilst in load/store instructions the destination register is given by bits 20-16. Therefore, we need to add an extra multiplexer to the `write_ addr` port of the registers. We also add ALU control, which takes the `func` field (bits 5-0) from the instruction and configures the ALU. This can be done completely separately to the rest of CPU control.

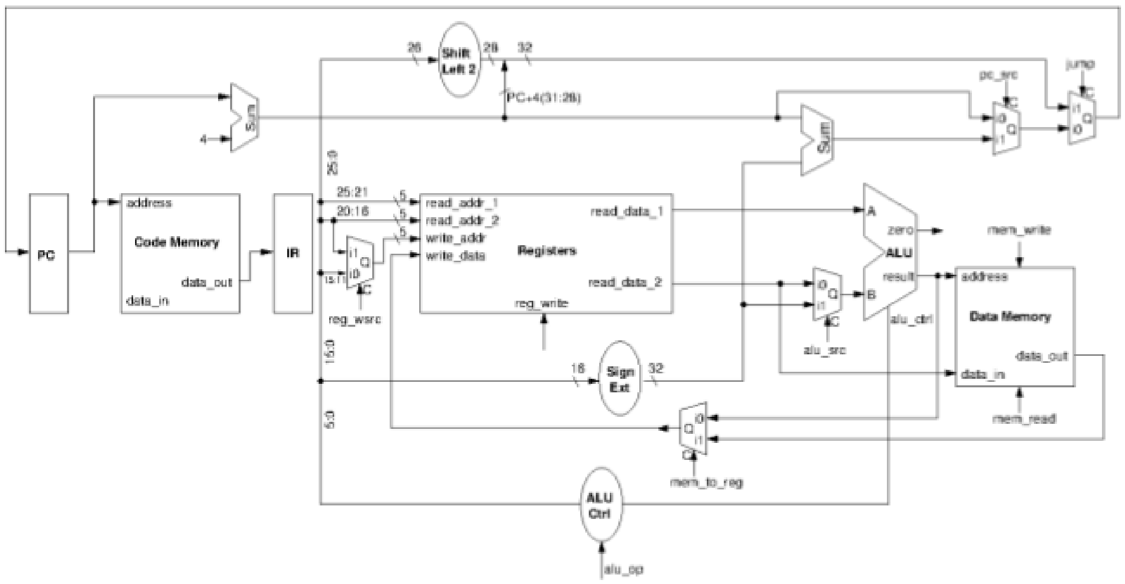


Figure 17: The complete MIPS datapath

Now, we have to address how to control this datapath so that it can be configured to execute each instruction. There are many control signals which must be set for each instruction. The information required to do this is encoded in the **opcode** and **func** (if appropriate) fields which must be read and interpreted to infer the correct configuration of the datapath that allows the data to reach the correct functional units. Furthermore, the ALU must be configured correctly and the memory configured to allow reads/writes if neccesary.

4.3 CPU Control

Control of a complex modern CPU is extremely difficult. We will start with the easiest aspect of CPU control: the ALU.

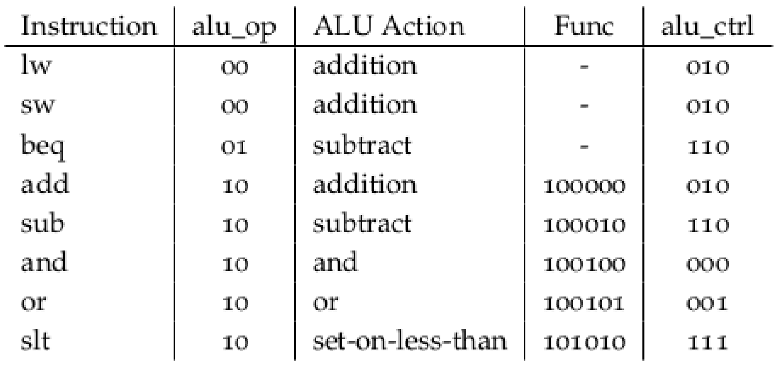
4.3.1 ALU Control

The ALU performs a large number of different operations, and the addition of a separate ALU controller simplifies the design of the control logic significantly. Note that for ALU instructions, the opcode is always `000000`, and the precise function is specified by the `func` code. There are other kinds of instructions for the ALU other than R-type instructions, so ALU control is not as simple as just reading the func code. For this, we have an additional control signal `alu_ op` which gives the ALU controller some extra information about what kind of instruction is being executed and whether `func` is needed or not.

Loads and Stores use the ALU to add the address offset to the base register, so it therefore must always be configured for addition. Branches on equality and inequality require the ALU to perform a comparisson. For R-type instructions, the ALU function is determined by the `func` instruction field. `alu_ op` is associated with each of these instruction types. The main control unit identifies the type of instruction from the opcode and sets the `alu_ op` control bits accordingly.

4.3.2 The Main Control Unit

Whilst the ALU controller handles the configuration of the ALU, it does so only in response to instructions from the main control unit. This unit is responsible for **decoding** the opcode of the current instruction to detemine what type of instruction it is, and configure the datapath



Instruction	alu_op	ALU Action	Func	alu_ctrl
lw	00	addition	-	010
sw	00	addition	-	010
beq	01	subtract	-	110
add	10	addition	100000	010
sub	10	subtract	100010	110
and	10	and	100100	000
or	10	or	100101	001
slt	10	set-on-less-than	101010	111

Figure 18: Summary of ALU control signals

(including ALU controller) appropriately. Whilst constructing the datapath, we have introduced a lot of control signals:

`reg_ write`	When true, allows writes to the register addressed by `write_ addr`
`alu_ src`	Selects whether ALU input is from register (False) or sign-extension unit (True)
`mem_ to_ reg`	Selects whether ALU output (False) or memory (True) is sent to registers
`reg_ wsrc`	Selects whether the register `write_ address` is bits [20-16] (True) or [15-11](False) of the instru
`pc_ src`	Selects whether PC+4 (False) or PC+4+branch (True) is sent to PC
`jump`	If true, sends jump address to PC; if false, sends the output of `pc_ src` to PC
`mem_ read`	If true, reading from memory is permitted
`mem_ write`	If true, writes to memory are permitted
`alu_ op`	Control bits for the ALU

5 Digital Logic

5.1 Digital Logic

5.1.1 Transistors

The basic building block of the modern integrated circuit CPU is the **transistor**. For the purpose of this course, we only need to know the transistors are essentially switches which are controlled by their gate voltage. Depending on the type of transistor (NMOS or PMOS), they will either conduct or not conduct when a voltage is applied to the gate. An NMOS transistor will conduct when the gate is at a positive voltage , and a PMOS transistor will conduct when the gate has no voltage - otherwise it doesn't conduct.

Figure 19: (Left) Circuit symbol for an NMOS transistor. (Right) Circuit symbol for a PMOS transistor

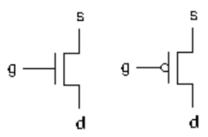
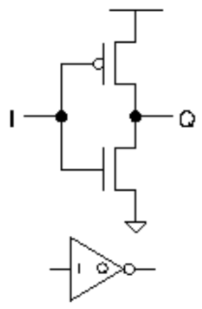


Figure 20: The logical operator **NOT** built from transistors



So far data and instructions have been represented by binary digits that have two possible states - zero or one. These states correspond to zero and positive voltages in physical circuits.

The zero voltage is usually called **gnd**(ground) and the positive voltage **vdd**. A signal that has a connection to gnd is in logic state 'zero', and a signal that has a connection to vdd is in logic state 'one'. Using transistors we can transform signals between 'zero' and 'one' states. To begin with, lets implement an inverter circuit which is identical to the logical operation **not** as shown in Figure 20.

When the input is at low voltage (zero), the lower NMOS transistor is non-conductive but the top PMOS transistor is conductive. The output therefore becomes connected through the PMOS transistor and is therefore logical 'one'. When the input is at high voltage (one), the PMOS transistor is non-conducting while the NMOS transistor is conducting. Q is therefore connected to gnd via the NMOS transistor and is logical 'zero'. We can see dependent on the input, the output of the circuit will be its inverse and it implements the logical **NOT** operator.

Similarly, we can implement other basic logical operators from a few transistors. While we might want to make **AND** and **OR** straight away, it's actually easier to implement their negated versions - **NAND** and **NOR**. We can do both of these in just four transistors, but it requires six to implement their non negated versions.

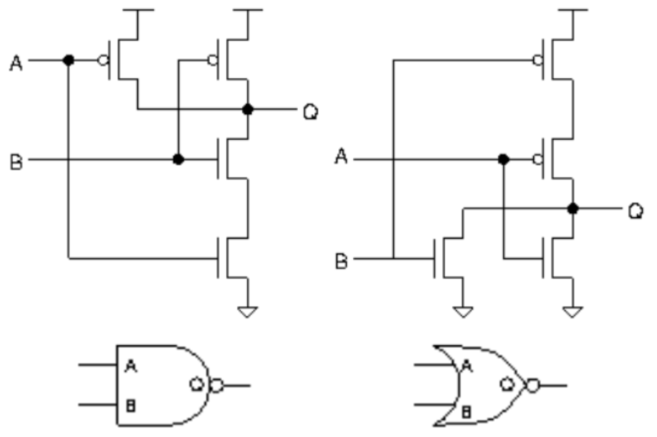


Figure 21: (Left) A **NAND** gate built from four transistors. (Right) A **NOR** gate built from four transistors

5.1.2 Decoders

Combinational logic is the general term for blocks of digital logic which contain no form of memory - the input must be determined solely by it inputs - and is made from networks of the simple logic gates we have just defined. Example of combinational logic we have seen whilst building the MIPS datapath include multiplexers, simple adders, and even an entire (non-pipelined) ALU.

Consider a simple block of combinational logic with two inputs and four outputs - a decoder. This is designed so that each of the four possible input combinations uniquely selects only one of the four outputs. This circuit is common in memories where it is used to select a unique memory location based on the presented address. We start by describing the truth table for this circuit as shown in the figure below. By inspecting the truth table, we can construct its function as a set of logic equations shown next to the truth table.

i0	i1	Q1	Q2	Q3	Q4
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Figure 22: Truth table for a two input decoder

- $Q_1 = \neg A \wedge \neg B$
- $Q_2 = \neg A \wedge B$
- $Q_3 = A \wedge \neg B$
- $Q_4 = A \wedge B$

From here it becomes trivial to translate this into a circuit diagram - you simply have to read off the logical operations (here, the mixtures of **NOT** and **AND**) and insert the appropriate gates.

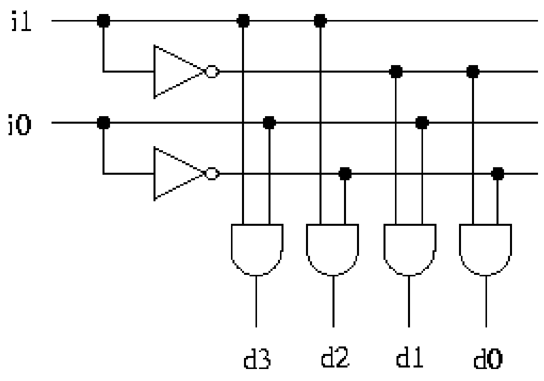


Figure 23: A 2-4 decoder built from logic gates

5.1.3 Multiplexers

We can also build **multiplexers** which are a very common component in the CPU. 2-1 Multiplexers are described by the logic equation $Q = (S \wedge I_1) \vee (\neg S \wedge I_0)$, and the truth table is shown below. Just as we did for a 2-4 decoder, we can easily build a 2-1 multiplexer from a few logic gates as described by its logic equation. In order to build 32-bit multiplexers, all we require are 32 single-bit multiplexers which share the same select signal 'S'.

i0	i1	S	Q
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Figure 24: Truth table for a 2-1 multiplexer

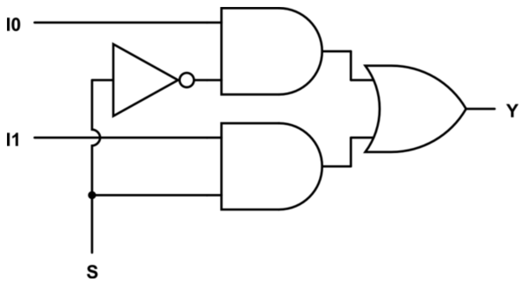


Figure 25: A 2-1 multiplexer built from logic gates

5.1.4 A Simple Adder

An **adder** is used both in the ALU and in several other places of the datapath. We will design a circuit which is able to add unsigned integers, two's-complement integers and both unsigned and two's-complement fixed-point numbers. For single digit binary addition, we know that $0 + 0 = 0$, $1 + 0 = 0 + 1 = 1$ and $1 + 1 = 0$ with a carry bit of 1. The carry bit of $1 + 1$ means we will need to introduce a second output which will function as our carry bit. It follows that we can easily express the carry bit as $C = A \wedge B$. In order to do the sum itself we need to introduce a new logical operation, **exclusive-OR** (otherwise known as **XOR**). **XOR** is indentical to and **OR** gate but does not output 1 if both input are also 1. Using and **XOR** we can fully implement single-bit addition. The circuit which achieves this is commonly called a **half-adder**.

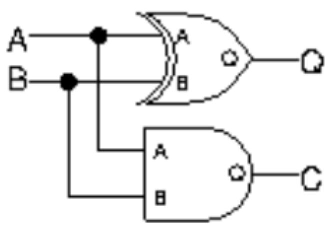


Figure 26: Logic gates which implemet a half-adder

This is clearly very limited as it can only add single bits together and we would like to add multiple-bit binary numbers. We can use a half-adder for the least-significant bit, but for higher bits we need to be able to carry in the the carry bit from the preceeding bit. We can easily achieve the sum using a pair of half-adders where the first adds I_1 to I_2, and the second adds the result of the previous to C_{in}. The carry bit of this addition is then the **XOR** of the carry bits from the two half-adders.

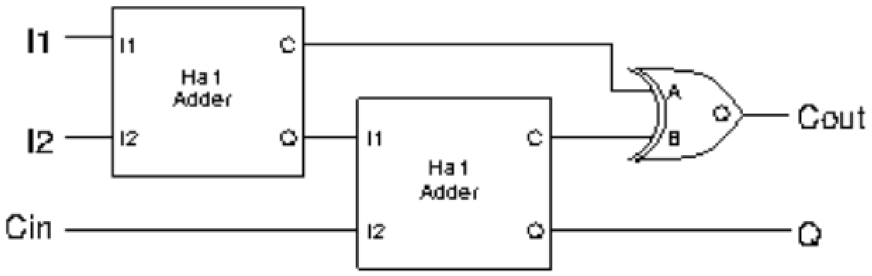


Figure 27: A full-adder built from two half adders and an **XOR** gate

We can now join lots of full adders (and one half-adder for the least-significant bit) together to create on single 32-bit adder. The carry bit from the most significant bit in the calculation will be used to indicate **overflow** (the result was too large to store in 32-bits).

5.1.5 A Simple ALU

Now we can build an adder, we can put together a simple ALU which can achieve addition of two numbers, bitwise-AND and bitwise-OR. To do this we take our 32-bit adder and modify it by adding one **AND** gate per bit and one **OR** bit per bit (with some multiplexers to select the operation we actually want to perform). We can then program the ALU by using the ALUOp signal to select the chosen function. How to build this ALU is shown in Figure 28. It is not too dificult to see how you could add additional functionality from here.

5.2 Number Representation

5.2.1 Positive Integers

The simplest of numbers we can represent in a binary computer system are positive (or unsigned) integers. This is the simple binary representation that we have already seen before and are common with. In principle, any natural positive number can be represented, but there are practical limitations. Modern computers break data into 32 or 64-bit chunks (referred to as words), which limits the size of numbers that we can store. With n bits, we can store integers from 0 to $2^n - 1$.

Long binary strings are tedious for humans to manipulate and mistakes are easily made. It's common to represent numbers in another base for this reason. Commonly, hexadecimal (base-16) is used. We need 16 symbols to represent values. We us the normal 10 decimal symbols, plus A-F. For example, the value of `0x1C3F` in decimal is:

$$(1 \times 16^3) + (12 \times 16^2) + (3 \times 16^1) + (15 \times 16^0) = 4096 + 3072 + 48 + 15 = 7231$$

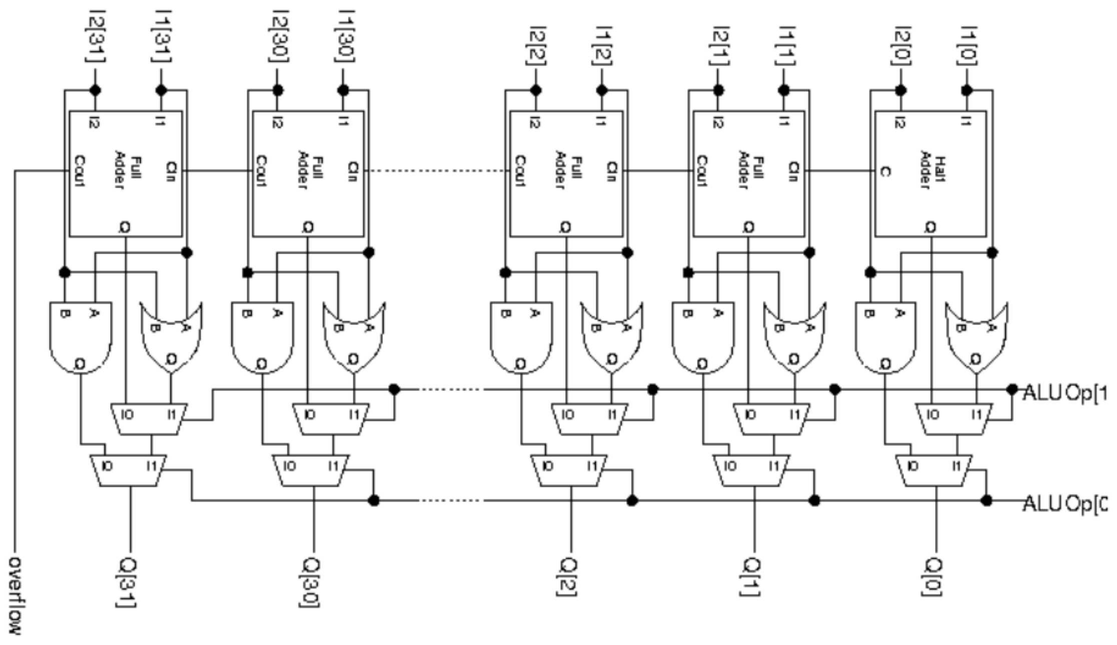


Figure 28: A simple ALU capable of performing addition, bitwise-AND and bitwise-OR operations

As 16 is an integer power of 2, converting between binary and hexadecimal is simply a matter of string substitution. Each group of four bits can be converted directly to their hex equivalent and vice-versa.

5.2.2 Positive Real Numbers

We can extend our representation of unsigned integers to unsigned reals (such as 2.1, 3.14 etc). We do so by allowing a decimal point in our binary number, and extending the powers backwards from 0.

$$
\begin{aligned}
101.001_2 &= (1 \times 2^2) + (1 \times 2^0) + (1 \times 2^{-3}) \\
&= 4 + 1 + \frac{1}{8} \\
&= 5\frac{1}{8}
\end{aligned}
$$

We have a choice to make, where do we put the decimal point? Modern computers have 32 or 64-bit words, but many real numbers are still impossible to represent with this. Dependent on the position of the decimal point, we can have an increased accuracy of numbers we can represent or a larger range of values. It's common practice to fix the decimal point at a known location, for example 16-bits for each side. This is called the **fixed point** representation .

5.2.3 Negative Numbers and Two's Complement

One possible way to implement negative numbers in binary is to allocate one of the bits as a sign bit. It makes sense to make this the leftmost bit, as the sign of a calculation cannot be determined until the whole calculation has been done. To do this, we either need an extra bit to represent out numbers or to reduce the range of numbers that we can represent. This method is not optimal for the computer. Take the following example: `01011010` and `10100110` are the negative of each other as they add to give 0 (ignoring any overflow). They are called each other's **two's complement**.

This is a convenient way of representing both positive and negative numbers. To form a numbers two's complement:

1. Invert all bits
2. Add 1 to the LSB

If we treat positive numbers to always begin with zero, and be otherwise identical to unsigned numbers then it follows that negative numbers must begin with at least one leading one. We'll consider the sum $(7-15)_{10}$ (which is equivalent to $(\texttt{00111} - \texttt{01111})_2$) and use two's complement to complete this. We convert $\texttt{01111}_2$ to it's two's complement and get $\texttt{10001}_2$. Now we can add to get $(\texttt{00111} + \texttt{10001} = \texttt{11000})_2$. As the lead digit is 1, this must be a negative number. We take its two's complement by inverting all bits, and adding 1 to get $\texttt{01000}_2$.

The range of numbers we can represent in two's complement is different than the number of bits we have. The largest positive value is `0111...11`, and the smallest positive value is `000...01`. For negative numbers, the smallest ('least negative') value we can represent is `111...11` whilst the largest ('most negative') is `100...00`. Therefore, the range of values that can be represented in two's complement is from -2^{n-1} to $2^{n-1}-1$

5.2.4 Floating Point Numbers

We can now represent both integers and real numbers (positive and negative) using **fixed-point** methods. These methods have restrictions. They impose considerable restrictions on the range of numbers that are available to use. Take for example an eight bit representation in two's complement using five bits to store the integer, and three bits to store the non-integer. This is bound by the following limits:

	Two's Complement	Radix 10
Largest Positive Value	`01111.111`	15.875
Smallest Positive Value	`00000.001`	0.125
Largest Negative Value	`10000.000`	-16
Smallest Negative Value	`11111.111`	-0.125

The range of values supported is rather small, and the precision available is also very limited. Fixed point data has a limited range and limited precision. We can circumvent this partially by allocating more bits, but there are practical limitations on this. To support more accurate numerical calculation, a **floating-point number** type has been devised. Our representation has two parts: the **mantissa** M (sometimes called argument, fraction or significand) and the **exponent** E which are combined to give a value V in the following way:

$$V = M \times 2^E$$

In 8-bits, allowing four bits of mantissa and four of exponent, and placing the radix point after the first bit of the mantissa we have (using two's complement for both exponent and mantissa):

	Floating Point	Radix 10
Largest Positive Value	$\texttt{0.111} \times 2^{\texttt{0111}}$	112
Smallest Positive Value	$\texttt{0.001} \times 2^{\texttt{1000}}$	4.9×10^{-4}
Largest Negative Value	$\texttt{1.000} \times 2^{\texttt{0111}}$	-128
Smallest Negative Value	$\texttt{1.111} \times 2^{\texttt{1000}}$	-4.9×10^{-4}

As it can be seen, there is a greatly increased range of values available from the exact same amount of bits. This comes at the cost of resolution. The use on an exponent means that the increments are non-linear, and hence larger numbers have a lower resolution than smaller numbers. As a result, you should be extremely careful when using floating point numbers.

Floating-point arithmetic is rather complex and the details are beyond the scope of this module.

5.3 Clocked Logic

Using combinational logic alone would make it very hard to build a full machine, as it takes different amounts of time for combinational circuits to stabilise, meaning when the input to a logic gate is changed it takes a finite amount of time for the output to change too. This time changes depending on the position of the chip, operating voltage, quality of materials and temperature. For these reasons, it's very difficult to know exactly how long to wait to ensure the correct answer. Asynchronous devices have been developed but these have only really been used as research devices so far and not deployed commercially. Almost all commercially available processors use an externally provided **clock signal** to ensure that the output of a functional unit is stable at a particular well-defined moment in time.

5.3.1 Clocks and flip-flops

To allow all signals to stabilise before they are used, we need to store them for the duration of the clock cycle. The start of a clock cycle is indicated by the **rising edge** of the clock signal. On this event, the outputs of the blocks of combinational logic are captured by special devices called **flip-flops**. The output of the flip-flop only changes on the rising clock edge, regardless of any changes on its inputs. Because of this, it is stable for the whole clock cycle which allows the outputs of any subsequent block of combinational logic to stabilise before the inputs change again. This is shown in Figure 29. Regardless of how the input changes at A, B remains stable for the entire clock cycle. This means that C has time to stabilise before the next cycle begins and transient states such as A4 do not propagate through the circuit at all. Such states can easily occur in complex logic as different paths through it may take different amounts of time and could lead to false results being produced.

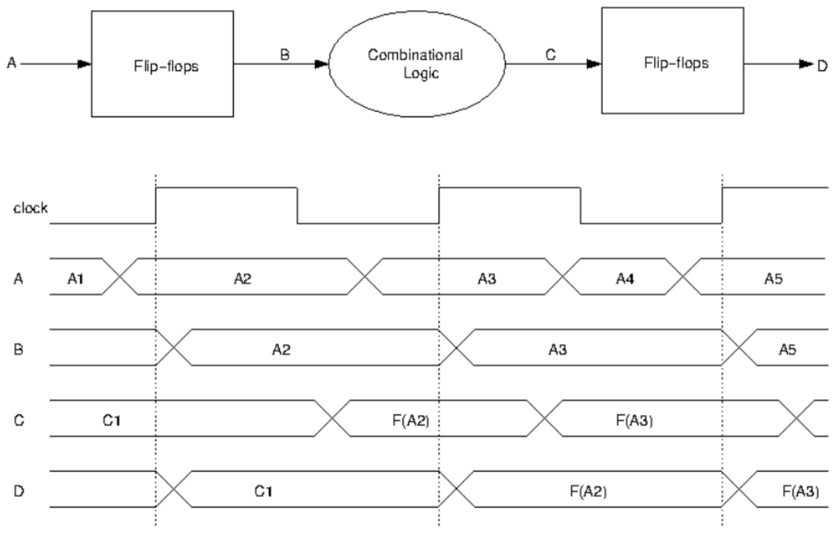


Figure 29: A timing diagram showing the effect of banks of ip-ops on the propagation of a signal through a circuit

Flips-Flops are widely used throughout the CPU wherever there is something to store or protect. In MIPS, each register is simply a bank of 32 flip-flops, and also an essential component of the pipeline.

5.3.2 The Master-Slave Flip-Flop

The most common type of flip-flop used in modern CPUs is the **master-slave** (or D-type) flip-flop. A basic D-type has two inputs, `datain` and `clock`, and one output `dataout`. Since logic gates are level sensitive not change senstive, it's not clear how to make a circuit that is edge-triggered. The key is to make it in two stages:

- The first stage allows the input data to pass only when the clock is low
- The second stage only allows data to pass when clock is high

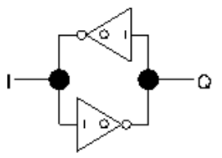


Figure 30: The most basic memory element, a pair of cross coupled inverters

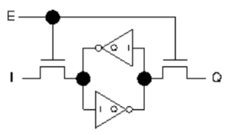


Figure 31: A six transistor SRAM cell. When the enable signal E is 0, the memory element is isolated

The basic building block is the **set-reset latch**, which implements a simple memory element. The simplest memory element is a pair of cross coupled inverters.

We need something with a little more functionality than the simple SRAM cell provides, and we use properties of NOR gates to achieve this. A NOR gate can be programmed to behave like an inverter by setting one of its inputs to 0, making the output the inverse of the other input.

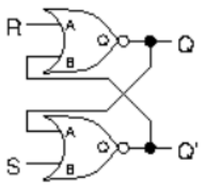


Figure 32: Cross coupled NOR gates form a bistable circuit

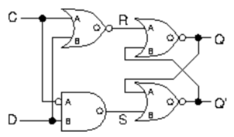


Figure 33: A modified bistable circuit forms a D-latch

- If R=0 and S=1, then Q=1, Q'=0 (setting the latch)
- If R=1 and S=0, then Q=0, Q'=1 (resetting the latch)
- If R=1 and S=1 then both outputs are 0
- If R and S are both zero, the outputs hold their current values

This component acts as a programmable memory element. In practice, we add extra logic to prevent R=S=1 and ensure the outputs are always the inverse of each other forming a **D-latch**.

- C=1 $\rightarrow$ R=S=0 regardless of the state of D: the latch is closed to the input and holds its current state.
- C=0 $\rightarrow$ R=$\neg$D and S=D
  - D=0 $\rightarrow$ S=0, R=1 $\rightarrow$ Q=0, Q'=1
  - D=1 $\rightarrow$ S=1, R=0 $\rightarrow$ Q=1, Q'=0
- S=R=1 can never happen

Whilst C=0, data on input D propagates through the circuit to output Q. The D-latch is therefore transparent whilst the clock is low. We combine two D-latches to form an edge-triggered flip-flop.

When the clock is low, the first D-latch (the master) is open and the output of this latch follows its input. The second latch is closed as it takes the inverse of the clock. When the clock is high, the second latch (the slave) is open and the first latch is closed. On the rising edge, the second latch captures the output of the first at that time, and prevents it from being changed until the next rising edge. We must therefore ensure that the input data is stable when the clock rises, otherwise the state after the change will be unpredictable. The output of the flip-flop can be read at any time, and is stable for the whole clock cycle expect for a very short period after the rising clock edge when it may change as the input is captured.

A very common modification is to add a mechanism so the flip-flop can be selectively written so we can use it in, for example, a register where we don't want the stored value to be overwritten

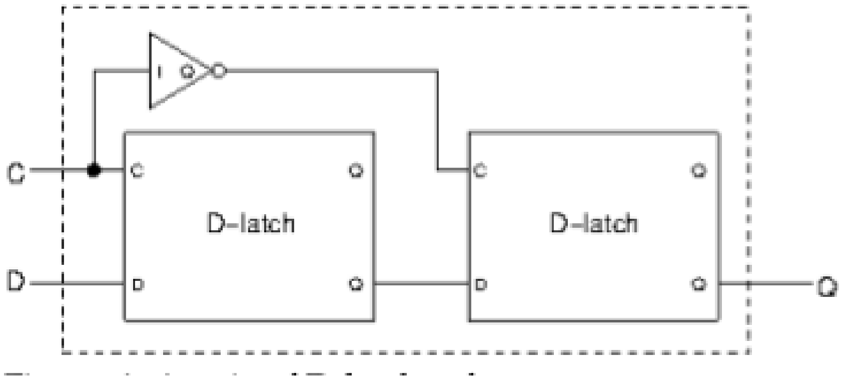


Figure 34: A pair of D-latches form an edge-triggered flip-flop

on every cycle. Usually, we feed the output back to the input, and add extra logic such that when write=1 the input of the master latch gets D and when write=0 the input of the master latch gets Q.

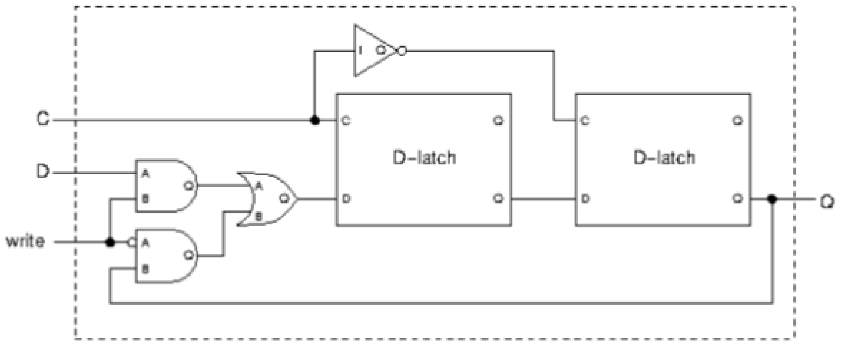


Figure 35: A write-enabled D-type flip-flop

5.3.3 Building a Register

The edge-triggered-flip-flop can be used as the basis for a register. A 32-bit register will simple be a bank of 32 D-type flip-flops organised such that the input to each flip-flop corresponds to one bit of the input data, and the output from each flip-flop corresponds to one bit of the output data. The clock and write signals are shared by all flip-flops, although we allow individual bytes to be written, in which case the register has separate write signals for each byte. This would be neccesary for MIPS, which has a store-byte (sb) instruction.

6 I/O and Peripherals

6.1 Input/Output Devices

Input and output are an essential part of computer systems. In most systems, I/O devices are not an integrated part of the main processor and memory and because of this we need to provide mechanisms that allow external devices to communicate with the processor, memory and the programs they execute. I/O has a few requirements:

- A communications channel between the processor and the I/O device (otherwise known as a **peripheral**).
- A way of identifying which I/O device we are communicating with.

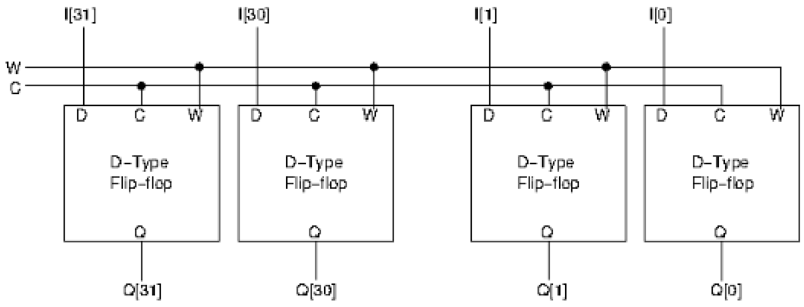


Figure 36: A write enabled D-type flop flop

- A mechanism by which we can determine whether the I/O device requires the processors attention, as I/O devices do not need constant attention from the processor due to their events occuring much less frequently than the processor clock speed.

The communications channel we need is achieved by the **bus**. The bus is a complex set of wires that carry signals which must be multiplexed between various devices and abide by various rules and protocols which determine which devices can send and recieve signals. Bus time is shared between all the system's devices and the bus controller will arbitrate between the devices to determine which can have access at any given moment.

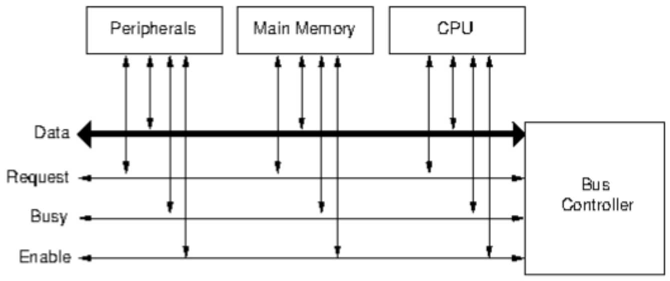


Figure 37: Schematic of the bus

6.2 I/O Models

The bus allows the CPU to communicate with peripherals in two ways: it can send them commands and it can send/recieve data. In principle, this is quite similar to how the CPU interacts with memory. In many ways, peripherals can be treated similarly to memory in what is known as **memory-mapped I/O**.

In this scheme, each I/O device is allocated a region of the address space (a block memory addresses). Each address in the memory will correspond to some form of register or buffer on the peripheral device. Memory mapped I/O is commonly employed in RISC designs as it allows I/O to be implemented with the minimum of addition of additional complexity since it is accessed by exactly the same mechanisms as the memory. MIPS is an example of a RISC design which uses this system. Addresses `0xFFFF0000` through `0xFFFFFFFF` are reserved for I/O device registers. This is a meager 64kB (16k words) of memory, a tiny amount of the total memory available but provides ample capacity to support I/O devices.

Alternatively, we could use **isolated** or **port-mapped I/O**. In this scheme, I/O devices have their own address space, instead of sharing the memory's address space. This is often done by adding a single control wire which indicates whether an address is for memory or I/O and requires special instructions which ensure that the correct address space is accessed. This comes at the cost of additional complexity in the form of specialised instructions for accessing the two different address spaces. This model is adopted by the x86 family of processors, which employs a logically separate I/O bus with its own address space.

Understanding how to control peripherals is just part of the problem of I/O. We also have to understand how to control the devices. For output devices, this is much easier. We simply send the appropriate instructions to the correct addresses or ports at the desired point in your code. We do need to check that the device is ready to accept the instructions before sending them.

For recieving inputs, the task is harder. Currently, we do not know when to check for data that has been sent to us, so we just do it constantly. The approach we have described so far is known as **programmed I/O**, and is sometimes is employed by microcontrollers. For multi-functional computers we need a less costly method of managing the I/O process - there are two means in common use to achieve this: **polling** and **interrupts**.

6.3 Polling

Take the example of reading keystrokes in from the keyboard as they are typed. We have no way of 'knowing' that there is a keystroke waiting to be read. We know the keyboard has a status register that tells us whether there is a value waiting in the keyboards data buffer. So far, we continuously checked the status register until there was data to be retrieved - and we have already noted how inefficient this is.

The keyboard generates data at a very slow rate compared to the processor, so it's not really neccesary to check the register so frequently. Instead, we test or **poll** the device regularly (but less frequently than we did before) allowing the CPU to perform other tasks between polling events. Polling requires no additional hardware and can be done entirely in software but may still waste a significant amount of processor time even if the polling frequency is reduced. If we reduce the polling frequency even further, we will help aliviate the problem further but we then run the risk of multiple events occurring between poll events and the earlier ones will be missed by the CPU. Polling gives us a simple software solution to I/O control, but it is by no means optimal.

6.4 Interrupts

Rather than repeatedly probing the device to see when it is ready as we do with polling, we would like a mechanism for the peripheral device to tell the CPU when it is ready - effectively allowing them to interrupt the current process. In this scheme, each device is allocated a special hardware **interrupt request channel** (IRQ) which can be used to signal the processor that it needs to pay some attention to the corresponding device. In between interruptions, the CPU is free to get on with other tasks without needing to repeatedly poll the device.

Interrupts require some specialist action by the programmer, who has to include a handler (subroutine) to deal with interrupt requests, and manually initiate I/O by sending appropraite commans to the relevant I/O controller. These routines are usually included in the device driver associated with a particular peripheral. Interrupt request channels are checked by the CPU each time a new instruction is fetched, to ensure that instructions are never left unfinished. If there is a pending request, the CPU acknowledges the request and disables any further interrupts. The CPU state is saved to the stack, and the interrupt handling subroutine is loaded. The state of the CPU is retrieved once the interrupt has been handled so that the main program's execution can continue. In this scheme, the CPU does not waste cycles polling the peripherals, so in this sense is a major upgrade. They are also far from perfect. The CPU is still responsible for serving the task, which may involve many millions of load/store operations in the case of a large data transfer between two hard disks. This may even block a more urgent request. We could solve this by prioritising devices and allowing a high priority device to interrupt a low priority device. Even better would be to pass control of memory intensive peripheral requests to a separate device that is dedicated to data transfer.

6.5 Direct Memory Access (DMA)

The DMA module is essentially a mini processor that specialises in transferring data between (virtual) memory locations which can include memory-mapped or isolated peripherals. When the

CPU needs to read or write a block of data, it issues commands to the DMA controller which consists of the address of the I/O device, the starting location of the data block, the number of words of data to be transferred and whether the transfer is read or write. The CPU can then return to the interrupted program leaving the DMA module to do the memory transfer via the system bus.

If the system has a cache, then care is required with the DMA. In particular, if the cache is write-back then we need to ensure that the correct version of the data is being copied. In this case, the cache would normally be **flushed** (any changed entries written back to memory) before the DMA transfer is allowed to start.

7 Improving Performance

7.1 Caches and Virtual Memory

So far, we have only spoken of memory as having two levels: **registers** and **the main memory**. This is a simplification, in reality most machine have at least two other levels - **cache** and **virtual memory**. The cache is a moderately fast, medium-sized memory which acts as a buffer between main memory and the registers.

7.1.1 Caches

Caches are a long established mechanism for improving machine performance and are usually used to store a copy of some part of the main memory. This leads to performance improvements because of two basic properties of computer programs:

- **The Principle of Spatial Locality**: This principle states that if you have recently referenced a particular item, you are likely to want to access nearby items soon. This is true of instructions because of their normally sequential execution sequence, and is true because memory is normally allocated in such a way that a program's data is stored in continguous memory locations.
- **The Principle of Temporal Locality**: This principle states that if you have recently referenced an item, you are likely to want to reference it again soon. The main constructs in software that leads to this are various types of loop.

Caches take advantage of these principles by storing instructions and data that are likely to be used again so taht they can be accessed more quickly. When a memory access is made, the cache is searched to see if it can be found there. If the item is in the cache, it is availble to the CPU much more quickly than if it were in main memory only. If the item is not in the cache, it is fetched from main memory into the cache and made available to the CPU. Over time, the contents of the cache stabilise and many fewer accesses to main memory are required as most memory requests can be serviced by the cache. Usually, separate caches are used for instructions and for data, since they are dealth with quite separately by the processor.

7.1.2 Structure and Design of Caches

Fundamentaly, caches are nothing more than a block of memory, but they have a few special features over and above main memory that are neccessary for them to perform their function. It is not sufficient for the cache to simply mirror the contents of the memory, the cache also needs to know whereabouts in the memory the item came from. For this reason, the cache must be able to store both the item that is to be cached and the address of that item in memory. Therefore, a cache consists fo two blocks of memory: one that stores the item itself and one storing the address of that item. In the cache, this is normally called the **tag**.

When a request is made to the cache, the memory address being accessed is compared to all the other addresses stored in the cache. If the address is in the cache, the corresponding item in the cache is read/written - a cache **hit**. Otherwise, we must access main memory - a cache **miss**. The addition of a cache to the system is therefore not free, it actually increases time taken to service a memory access in the case of a cache miss.

7.1.3 Cache Associativity

One of the biggest factors affecting the performance of a cache is its degree of associativity. This is the number of different locations that an item of data can occupy in the cache.

One of the most common types of cache is the **direct-mapped**, which is one-way associative. In the design, data from each address in main memory can go in only one location in the cache. This makes the cache easy to design and fast (in terms of access times). A subset of the address is hard-coded into the cache so that when an address `101x` is presented, it is only neccesary to look at one cache entry. The remaining address bits are checked against the tag for that entry, and if they match, this is a cache hit and the corresponding entry in the data array is accessed. The largest problem with this approach is that we cannot store items with addresses `111x` and `111y` in the cache at the same time. The potential consequence of this is known as **thrashing**, where entries in the cache are continually swapped which destroys any performance benefit that the cache may have brought. Careful choices of the hard-wired address bit can minimise this, but sometimes this may not be enough.

One solution is to allow each piece of data to be stored in multiple locations in the cache. In an **n-way associative cache**, each address in main memory maps to n cache entries. A smaller portion of the memory address is hardcoded in to the cache, and the memory address then has to be compared with n entries in the tag.

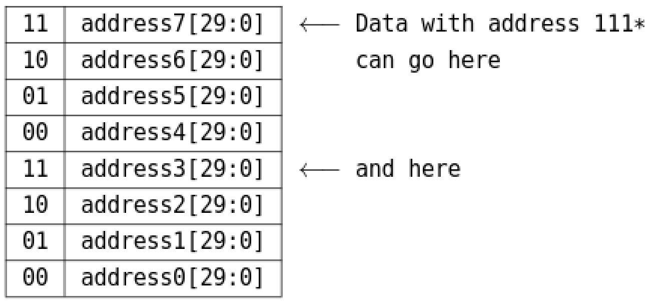


Figure 38: Address mapping in a 2-way set associative cache

If we set n to be the same as the number of locations in the cache, we say the cache is **fully associative**. There is no hardwired address mapping, and any item can go anywhere in the cache. All tag entries have to be checked for the desired address which increases complexity, but it can greatly reduce the potential for thrashing.

7.1.4 Refill Strategies

Following a cache miss, the requested item is retrieved from memory and should then be stored in the cache as it is likely to be used again soon. In a direct-mapped cache, there is only one location where this new item can go. In an n-way associative cache, there are n possible locations that we could store the item so we need to decide which one should be replaced. This is governed by the **refill strategy**.

The simplest strategy is to replace the item that has been least recently used (LRU) on the grounds that it is less likely to be needed soon. This may cause problems in loops, especially when the loop body is larger than the cache size. This method also requires a lot of extra housekeeping if n is large.

Another strategy is to simply randomly replace, which does not use any of the principles of how the cache operates, but is far easier to implement. It will greatly increase miss rate, so the designer must choose whether they increased complexity or increased miss rate is most acceptable.

7.1.5 Performance Benefits of Caches

When calculating the performance benefit, we make the following assumptions:

- All instructions can be fetched in a single cycle (removing the need for an instruction cache).
- All instructions execute in a single cycle, apart from those which access memory
- There are N instructions in the program
- A proportion m of the instructions access memory
- The cycle time is given by t_{cyc}.
- The memory access time (additive to the cycle time) is given by t_{mem}.
- The cache access time is given by t_{cache}.
- The cache hit rate is h.

Using these parameters, the time taken to execute the program consists of three parts:

- The time taken to load all instructions (and execute non-memory instructions, which is done simultaneously with the next fetch): $N \times t_{cyc}$
- The time taken to access the cahce for memory instructions: $N \times m \times t_{cache}$.
- The time taken to access main memory when the cache misses: $N \times m \times (1-h) \times t_{mem}$.

Therefore, the total program execution time is

$$\begin{aligned} T &= N \times t_{cyc} + N \times m \times t_{cache} + N \times m \times (1-h) \times t_{mem} \\ &= N(t_{cyc} + m(t_{cache} + (1-h)t_{mem})) \\ &= N\langle t \rangle \end{aligned}$$

where $\langle t \rangle$ is the average time to execute an instruction.

Consider a program running on a machine with 1ns cycle times (1GHz clock rate) and a memory access time of 10ns. The program takes 6s to run, and 20% of its instructions access memory. A cache is then added with an expected hit rate of 80%, and an access time of 5ns. We will calculate the time of execution of the program with the cache. Without the cache, we have that the average time to complete an instruction is:

$$\begin{aligned} \langle t_1 \rangle &= t_{cyc} + m \times t_{mem} \\ &= \text{1ns} = 0.2 \times \text{10ns} \\ &= 3ns \end{aligned}$$

Now lets introduce the cache:

$$\begin{aligned} \langle t_2 \rangle &= t_{cyc} + m(t_{cache} + (1-h)t_{mem}) \\ &= \text{1ns} + 0.2 \times (\text{5ns} + 0.2 \times \text{10ns}) \\ &= \text{1ns} + 0.2 \times \text{7ns} \\ &= \text{2.4ns} \end{aligned}$$

Therefore, the cache has reduced the total execution time of the program from 6s to 4.8s, a 20% reduction in program execution time.

7.1.6 Writing Strategies

Read access to caches are easy: if the data is in the cache, get it from there, otherwise go out to main memory and get it from there. Write access raise some potential difficulties. Foremost is whether we should update the cache and memory simultaneously. An important issue is the **coherance** of caches and the memory.

In a **write-back** cache, only the cache (not the memory) is updated during a write hit. This reduces the number of memory accesses, especially when a variable is updated frequently, but we must make sure that main memory is updated when this entry in the cache is replaced. For this, we must have a write flag in the cache that informs us something has changed and the memory is out of date.

In a **write-through** cache, both main memory and the cache are updated simultaneously. This ensures that memory and cache are consistent (coherent) but requires extra memory accesses.

The choice of write policy depends on the system. In a system with a single CPU core accessing memory, write-back is probably a safe strategy. If other devices such as DMA modules are allowed to talk directly with memory, then care is required to ensure that memory is kept up to date. This means these devices neeed to either be routed through the cache or the cache must be flushed such the updated words are written back to memory.

7.2 Pipelining and Branch Prediction

We have studied caches and their performance benefits, but there are a number of other methods which are used to

- Improve the processors throughput of instructions - **pipelining**
- Execute more than one instructions at a time - **superscalar processors**
- Predict what the machine will do next - **branch prediction**

7.2.1 Pipelining

The basic idea behind pipelining is that we don't wait for an instruction to finish executing before we start the next instruction. Imagine a factory: production lines break the manufacturing process into a series of sub-stages. There may be many items on the line simulataneously, each at a different stage of production. Each item may take the same amount of time to produce, but throughput is much higher.

Implementation of a pipeline is a complex process. Its essence is to split the datapath into pieces, each piece responsible for one portion of the pipeline. Pipeline registers are added between pipeline stages to store intermediate results, and during each cycle the processor only needs to push an instruction through one pipeline stage. This allows the CPU to run at a much higher clock speed.

We may break down MIPS instructions to a five stage pipeline:

1. **IF (Instruction Fetch)**: Fetch instruction from memory
2. **ID (Instruction Decode)**: Decode instruction and read registers (can happen simultaneously in MIPS)
3. **EX (EXecute)**: Execute the operation or calculate memory address
4. **MEM (MEMory)**: Access memory if required
5. **WB (Write Back)**: Write result into a register if required.

This is often referred to as the class five-stage RISC pipeline due to its use in several common 1990s RISC CPUs. It was designed to in principle allow the CPU to complete one instruction per cycle, with each pipeline stage working on a different instruction.

There are some situations where pipeling can cause significant problems. Consider the following:

```
addi $t0, $t0, $t0; // add 1 to t0
add $t0, $t0, $t0;  // double t0
```

Here, the result of the first instruction must be known before the second can be done, so we must wait until the `addi` has been written to registers before the ALU stage of the add instruction can complete. We must delay the start of the `add` by adding a **bubble** into the pipeline as shown in Figure 39. These situations may be spotted by the compiler at compile-time or by the processer in a pre-processing step.

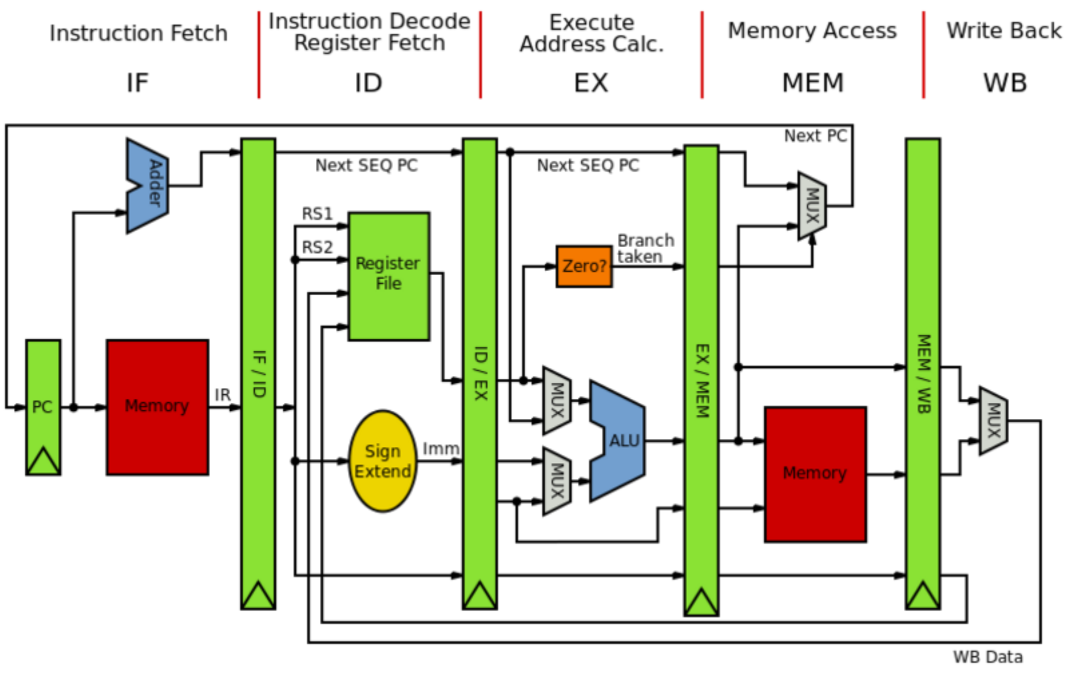


Figure 39: The MIPS five stage pipeline

addi	IF	ID	EX	MEM	WB			
add				IF	ID	EX	MEM	WB

Figure 40: Execution is delayed by two cycles until the results of the `addi` instruction are available

Another way of dealing with this is to make the result of the ALU stage of the pipeline available immediately by adding a physical short-cut. This is known as **forwarding** or **bypassing**. In the pipelined MIPS system this would be from the output of the EX/MEM pipeline register to the input of the EX pipeline stage

7.2.2 Branch Prediction

When we have a branch, we must know the result of the comparison before we know which instruction we should be executing next. If we can accurately predict what should happen next, we can get a significant performance boost - particularly if the branch instruction is executed often as is the case with loops. We have two possible instructions to execute next. If the branch is not taken, the instruction we want is the next in the program sequence. If the branch is taken, we must compute the branch address before we known which instruction we must execute. If a good guess can be made as to which branch will be taken, one of the branches can be speculatively executed to good effect.

There are two simple choice we could make. Firstly, we could assume that all branches are not taken, and allow the CPU to just execute the next instruction in the code before the results of the branch comparison are known. If the branch isn't taken, carry on. If the branch is taken, we have to flush the pipeline of incorrect instructions so the the correct instructions can be executed.

On the other hand, we could assume branches are always taken. This isn't as simple, as the branch instruction involves both a comparison and the computation of the branch address. Due to this, the address of the next instruction is not known until after this has been done. If we add specific hardware to computing branch addresses, this can usually be done with only one cycle of delay to the pipeline, which will require a bubble in the pipeline. As before, if the branch is not taken, we must flush the pipeline and execute the correct instruction.

We can improve by looking at the context in which a branch is used. For example, a branch which breaks out of a loop when the termination condition is met is far more likely to be taken than it is to not be taken. Similarly, in a long chain of if-elseif statements it has been shown that branches are taken more often than they are not taken. Accurate predictions can also be made on the opcode alone: perhaps a branch-on-equal is much less likely to be taken than a branch-on-less-than-or-equal. Making accurate decisions requires careful analysis of the behaviour of programs and also compilers. This is usually referred to as **static branch predicition**.

A more **dynamic branch prediction** methodology can give even more benfits at the cost of resources. One common approach is for the CPU to keep a history of each branch and use it to predict future behaviour. These methods can be up to 90% accurate but introduce significant design complexity.

Another approach is to use delayed branching, where code is reordered so that the branch is executed out-of-sequence and earlier than would normally be the case so that the result of the branch comparison can be known in time to determine the correct branch to take at the correct time. This works well provided the branch comparison does not depend on the result of a preceeding instruction.

7.2.3 Superscalar Processors

In recent years, we have been able to put more and more functional units on a chip. Superscalar machines have multiple pipelines which can execute multiple instructions in parallel. With N pipelines, performance is increased by a factor of N. This comes at the cost of making control very difficult. Especially difficult is ensuring correct instruction ordering as we cannot execute instructions in parallel if they ae co-dependent.

Superscalar techniques differ from modern multi-core methods (which also use superscalar techniques within each core) as they are designed got instruction-level parallelism within a single executing process. The two techniques can happily coexist though.

List of Figures

Index