The Moncky project

Kris Demuynck

18th August 2021

Abstract

This document describes the architecture and implementation of the Moncky processors and the Moncky computer. It is assumed that the reader is familiar with logic circuits, processors, and Verilog. This document is a work in progress and will be updated regularly. The home of the Moncky project is https://gitlab.com/big-bat/moncky.

Copyright

Kris Demuynck

The Moncky processor family

Contents

Chapter 1

Introduction

The Moncky-1 processor was originally conceived to accommodate students in learning and experimenting with processors and assembly language. It is a 16-bit processor that has a single cycle RISC architecture with a minimal instruction set (only 7 instructions). It does not have interrupts and has a Harvard architecture. This processor cannot be implemented in an FPGA as it contains busses and 3-state buffers (which are not available on an FPGA). It is implemented as a simulation in Logisim (http://www.cburch.com/logisim/) and Digital (https://github.com/hneemann/Digital). The course text that describes logic circuits and the Moncky-1 processor can be found at:

`https://drive.google.com/file/d/1ngHiIrxpOfb0dO7pm3ZOe7q8XEXCb-b6/view?usp=sharing`

The Moncky-2 processor is a synthesizable version of the Moncky-1. The architecture was changed significantly with extra instructions in mind (these are available in the Moncky-3 processor). There is a simulation available in Digital.

The Moncky-3 processor is an extension of the Moncky-1 and 2 instruction set. It still executes every instruction in one clock cycle but has instructions that can do more things at once. The Moncky-3 processor also has an interrupt line so external hardware can send an interrupt which will stop the processor and make it execute a specific interrupt handler.

The Moncky-3 computer is a full blown computer containing the Moncky-3 processor, memory, ROM, video output, keyboard input, and SPI.

The Moncky assembler is a java program that can translate Moncky assembly language to machine code. It supports all Moncky processors.

Chapter 2

The Moncky-1 processor

In this chapter a complete working processor (the 'Moncky-1') is presented. It is a fictive processor that is specifically designed for educational purposes. Although real processors are much more complicated it is capable of performing the same tasks, except for tasks that involve interrupts. It is perfectly feasible to create a physical chip for the Moncky-1, using ASIC technology. Creating an implementation on an FPGA is not possible as there are multiple busses and tri-state buffers which are not available in most FPGA's (see also the Moncky-2 processor). A simulation in Logisim and Digital exists and can be found at gitlab:

`https://gitlab.com/big-bat/moncky`

The Moncky-1 processor has the following properties:

- it has a 16 bit architecture: all registers and all instructions are 16 bits wide
- there are only 7 instructions
- each instruction can be performed in one clock cycle
- memory is limited to 2x128 KiB
- there are no interrupts
- it has a Harvard architecture

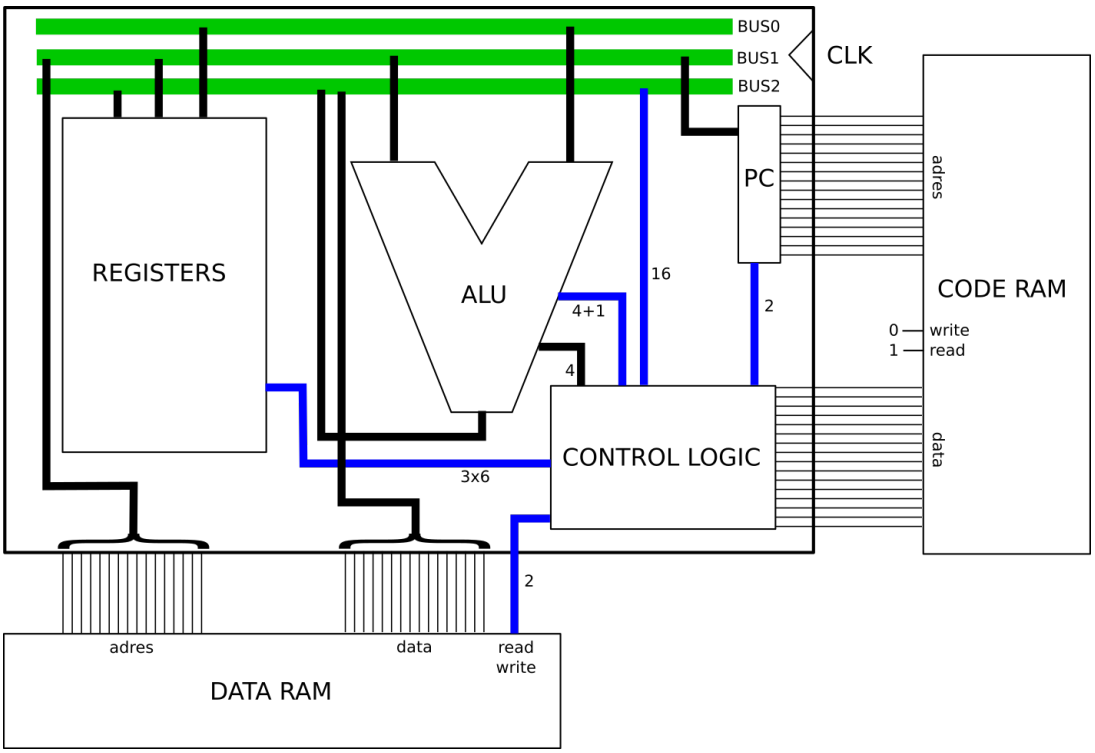
2.1 Architecture

The Moncky-1 processor contains the following components:

- the 'registers': these hold temporary data. There are 16 registers which are all 16 bits wide.
- the 'PC' register which contains the address of the current instruction
- the 'ALU': this will perform arithmetic and logic operations on the data stored in the registers
- the 'instruction decoder' or 'control logic': this interprets the instructions and controls the other components in the processor

- internal busses: these connect all the components together

The Moncky-1 processor has the following architecture:



The black lines in this diagram are data lines and the blue ones are control lines.

Two RAMs are connected to the processor (Harvard architecture). Both RAMs have an address bus that is 16 bits wide. The data bus is also 16 bits wide. A RAM can thus be regarded as an array of 65536 values. Each value is a 16 bit number. The RAMs are assumed to be built with D-latches and to be asynchronous which means they don't have a clock. Reading from the RAM is done continuously as the address bus is directly connected to the MUX in the RAM.

The 'code RAM' contains instructions that need to be executed. Each instruction is 16 bits wide. The 'data RAM' is a memory that will hold the values of variables. Using two RAMs makes the design of the processor simple and keeps performance high.

There are 16 registers in the Moncky-1. Each register can hold a number of 16 bits. The registers are numbered: r0, r1, ..., r15. They are used to hold temporary data i.e. data that is used to perform a calculation. If data needs to be stored, it is always saved to the data RAM.

The program counter (PC) is a separate register directly connected to the address bus of the code RAM. It is a counter that starts at 0 and increments each clock cycle. It holds the address of the instruction that needs to be executed. The code RAM is hardwired for reading. It will put the value at the given address on its data bus. The register PC is sometimes called 'instruction pointer' (IP) as it points to the current instruction. In practice the code RAM can be implemented as an EEPROM that is programmed by external hardware.

The ALU (Arithmetic Logic Unit) is capable of doing calculations. It reads two numbers, performs the calculation, and puts the result on its output. It also generates 'flags' that give information about the result. The inputs are on top and the output is on the bottom.

There are 3 busses in the Moncky-1. They allow the components to send information to each other. For example, the ALU will read the two numbers from busses 0 and 1. The value on bus

0 and bus 1 will come from a chosen register. The control logic will determine which registers are put on the busses. The result of the ALU is put on bus 2 and can be read into a register.

The control logic is connected to the data bus of the code RAM. It will read the instruction and will translate this into control signals to the other components.

The processor has a clock input. All registers, including PC, share are connected to it. They respond to the falling edge of the clock.

2.2 The instruction set

This section describes the instruction set architecture (ISA) of the Moncky-1 processor.

In the Moncky-1 processor all instructions are exactly 16 bits long. It knows the following 7 instructions:

- stop executing instructions (this is used to stop the processor after executing a program)
- load a number (a constant) into a register
- perform a calculation with the registers and put the result back into a register
- load a number from data RAM (at a given location) into a register
- store a number from a register into data RAM at a given location
- jump to a specific address in the code RAM
- jump to a specific address in the code RAM if the result of the previous calculation was zero, negative, positive, ...

Every instruction is embedded in 16 bits. These bit patterns constitute the 'machine language' of the processor. Because they are very hard to read, a more human readable version is also provided. It consists of a short command name (opcode) and parameters (operands). This language is called the Moncky-1 'assembly language'.

In the Moncky-1 processor the 7 instructions are encoded in 16 bits as follows:

machine language	meaning	example
`0000 xxxx xxxx xxxx`	the processor stops	halt
`00x1 iiii iiii rrrr`	R[r] = 0000 0000 iiii iiii	li r3, 42
`01xx AAAA rrrr ssss`	R[r] = R[r] ALU R[s]	add r3, r4
`100x xxxx rrrr ssss`	R[r] = RAM[s]	ld r2, (r1)
`101x xxxx rrrr ssss`	RAM[s] = R[r]	st r1, (r2)
`110x xxxx xxxx rrrr`	PC = R[r]	jp r12
`1111 xxxx xccc rrrr`	if (condition) PC=R[r]	jpz r1

The following notation is used:

- R represents the array of registers
- 'iiii iiii' stands for an 8-bit value that is embedded into the instruction. This is also called an 'immediate' value as the value is immediately available from the instruction.
- "rrrr" and "ssss" stand for register numbers (0-15)

- "AAAA" stands for an ALU operation. The following operations are supported:
  - 0000 = NOP (No OPeration: the result is equal to the second input of the ALU)
  - 0001 = OR (a bitwise OR)
  - 0010 = AND (a bitwise AND)
  - 0011 = XOR (a bitwise XOR)
  - 0100 = ADD (an addition)
  - 0101 = SUB (a subtraction)
  - 0110 = SHL (shift left: the first number is shifted to the left. The number of bits to shift is indicated by the second number)
  - 0111 = SHR (shift right)
  - 1000 = ASHR (arithmetic shift right)
  - 1001 = NOT (logic inverse: the result is the 1-complement of the second input)
  - 1010 = NEG (negative: the result is the 2-complement of the second input)
- "ccc" indicates a certain condition. Every time a calculation is done by the ALU, 4 flags are set according to the result: the carry flag, the zero flag, the sign flag, and the overflow flag). These flags can be used to decide to jump to a certain location in the code RAM. The following conditions can be checked:
  - 000 = 'c' = if the carry flag is equal to 1 (the calculation resulted in 1 extra bit being discarded)
  - 001 = 'nc' = if the carry flag is equal to 0 (the calculation did not result in an extra bit)
  - 010 = 'z' = if the zero flag is equal to 1 (the result of the calculation was 0)
  - 011 = 'nz' = if the zero flag is equal to 0 (the result of the calculation was not equal to 0)
  - 100 = 's' = if the sign flag is equal to 1 (the result of the calculation was negative)
  - 101 = 'ns' = if the sign flag is equal to 0 (the result of the calculation was not negative)
  - 110 = 'o' = if the overflow flag is equal to 1 (the calculation resulted in an overflow)
  - 111 = 'no' = if the overflow flag is equal to 0 (the calculation did not result in an overflow)
- "x" stands for don't care (the values of these bits can be anything; we will always put them to zero in the examples below)

The 4 most significant bits indicate which instruction is to be executed. They are called the 'opcode'. The other bits hold the parameters for that instruction. These are called the 'operands'.

A small program for the Moncky-1 could look like the following:

```
;machine code         assembly    meaning
0001 0000 0101 0000   li r0, 5    ; load the number 5 in register 0
0001 0000 0011 0001   li r1, 3    ; load the number 3 in register 1
0100 0100 0000 0001   add r0, r1  ; add register 0 and 1 together
                                    and put the result in register 0
0001 0000 0010 0001   li r1, 2    ; load the number 2 in register 1
1010 0000 0000 0001   st r0, (r1) ; store the value van r0 (8 in this case)
                                  ; at location given by r1 (2 in this case)
                                  ; in the data RAM
0000 0000 0000 0000   halt        ; end the program
```

The left column shows the binary form of the machine code. These are the bits that are stored in the code RAM. The middle column shows the assembly version of the instructions. The third column contains comments (here a comment starts with a semicolon).

This program is stored in the code RAM. This means the code RAM will contain the following values (in hexadecimal): 1050, 1031, 4401, 1021, A001, 0000.

When the processor is started, PC is set to 0. This means that the first instruction will be executed. After one clock cycle PC is incremented and the next instruction can be executed. This continues until PC=5. Since the instruction is 'halt', PC will not be incremented any more. This halts the processor.

2.3 The internal components

In this section the different components of the Moncky-1 processor are discussed in more detail. There is a simulation in Logisim and Digital available via:

```
https://gitlab.com/big-bat/moncky
```

2.3.1 The busses

The Moncky-1 contains 3 busses that connect the data inputs and outputs of the different components. They are numbered 0, 1, and 2. Every bus consists of 16 lines. The registers, the ALU, the PC register, the data RAM and the control logic are connected to them using tri-state buffers. The control logic is the master of the busses and decides who can write to which bus.

2.3.2 The registers

The Moncky-1 processor contains 16 registers of which each represents a 16-bit number. All registers are built with master-slave D flip-flops. They all share the same clock. At the falling edge of each clock cycle a register can read a new value from a bus of write its value to a bus. MUX's and DEMUX's are used to select the right register and to indicate if it needs to read, write, or just retain its value.

Looking at the registers as one big sequential circuit, it has the following connections:

- a clock input that is forwarded to all 256 flip-flops.
- per bus:
  - 16 connections that can be configured as input or output. These are used to send or retrieve data
  - 4 inputs that determine which register should be connected to the bus (they lead to the MUX's and DEMUX's). These come from the control logic.
  - 1 input (rw) that determines if the the selected register should read or write. Its value is determined by the control logic.
  - 1 input (enable) that connects the selected register to the bus, or disconnects it. This will drive the tri-state buffers. Its value is determined by the control logic.

For example, if the control logic wants to send the information on bus 0 to register r5, the 4 inputs for bus 0 will be put on 0101. The rw input will be 0 (read) and the enable input will be 1. The enable inputs of the other busses will be set to 0 and their rw will be put at 1 (write). When the clock now goes from 1 to 0, register r5 will read the value from bus 0.

In fact the registers form a triple port RAM of 16 words. Each port is connected to a bus.

The PC register is also a register, but it can also increase its value by 1 every clock cycle. The PC register can put its value on bus 1 or read a new value from it.

2.3.3 The ALU

The ALU is in fact a very big combinational circuit. It has 37 inputs and 20 outputs. The output is always directly determined by the input. The inputs are:

- 16 inputs for the first number
- 16 inputs for the second number
- 1 input that determines if the output of the ALU should be connected to bus 2
- 4 inputs to select the operation the ALU needs to perform.

The outputs are:

- 16 outputs for the result
- 4 outputs that give information about the result (the flags):
  - zero flag: is the result equal to 0?
  - carry flag: is the most significant carry bit set?
  - sign flag: is the result negative?
  - overflow flag: was there a 2-complement overflow?

2.3.4 The control logic

The control logic is a main component in the processor. It interprets an instruction and drives all other components. It is a big combinational circuit with 20 inputs and 42 outputs.

The inputs are:

- 16 bits for the instruction. These bits come from the code RAM.
- 4 bits coming from the ALU (the flags)

The outputs are:

- 2 signals go to the control bus of the data RAM and indicate if the RAM needs to read or write or do nothing
- 4 signals go to the ALU to indicate the operation it needs to perform
- 1 signal goes to the ALU to indicate if it has to put its output on bus 2

- 2 signals to the PC register to determine if the register should increment, read its value from bus 1, or write its value to bus 1
- 3x6 signals to the registers determining which register should be connected to each bus and what action it should take
- 16 data signals to bus 2 to send a number directly to a register

The control logic consists of 7 circuits that drive their respective instruction. A MUX is then used to select the right control signals based on the opcode of the instruction.

The Moncky-1 processor is designed so that every instruction in executed in one clock cycle. When the clock is low, the control logic interprets the instruction and puts all the control signals to the right values. When the clock goes high nothing changes. The ALU and the data RAM get time to perform their operation and will put the result on the right bus. When the clock goes from one to zero (falling edge), the registers store the new value and the program counter is increased. When PC changes, a new instruction is presented to the control logic. The next subsections will discuss a few instructions and explain in detail how the control logic drives all components in order to execute the instruction.

2.3.4.1 Load instruction (ld)

The first instruction that will be discussed is 'ld r2, (r1)'. In this instruction r1 points to a location in the data RAM. The processor needs to load the value at that location and store it in register r2. Let's assume that register r1 contains the value 1024 and that the instruction itself is stored at location 2040 in the code RAM.

At the falling edge, when the clock goes from 1 to 0, PC reaches the value 2040 (`0000 0111 1111 1000`). This value is put on the address bus of the code RAM which drives the MUX within it, selecting the right value at this location. The RAM will respond by putting the instruction (`1000 0000 0010 0001`) on its data bus. The control logic notices this instruction begins with '`100`', so it knows it is a 'ld' instruction. The following signals are then sent to all components:

- the registers get the following signals per bus:
  - bus0: is decoupled (enable signal is put to 0)
  - bus1: bits 0-3 of the instruction (`0001`) are forwarded so that register r1 is selected. The rw signal is put to 0 (read) and the enable signal is put to 1. As a result, register r1 will put its value on bus1
  - bus2: bits 4-7 of the instruction (`0010`) are forwarded so that register r2 is selected. The rw signal is put to 1 (write) and the enable signal is put to 1. As a result, register 2 will read its value from bus2
- the data RAM is configured for reading (read=1, write=0). It will put the data on bus2.
- register PC is asked to increment itself on the next clock cycle

As a consequence of these signals, the value of register r1 (1024) flows through bus1 to the address bus of the data RAM. The data RAM will put the value stored at this location on bus2. The value is read by register r2. This register contains flip-flops so it does not yet store the new value.

When the clock goes from 1 to 0, register 2 will store the new value and PC will increment. The instruction is now executed and, since PC is incremented, the next instruction will appear in the control logic.

2.3.4.2 A calculation (add)

The second instruction is 'add r2, r3'. This instruction adds the values of r2 and r3 and puts the result back into r2. The instruction looks like: `0100 0100 0010 0011`.

When the control logic gets this instruction, it notices it begins with '01'. The following signals are now sent:

- bits 8-11 of the instruction contain the operation that needs to be performed ('add' in this case). They are sent to the ALU.
- the ALU is instructed to put its value on bus2 (using tri-state buffers)
- the registers get the following signals:
  - bus0: register r2 puts its value on this bus
  - bus1: register r3 puts its value on this bus
  - bus2: register r2 reads its value on this bus
- register PC is instructed to increment

Notice that register r2 puts its value on bus0 and reads a new value from bus2. This is possible due to the master-slave architecture of the flip-flops.

The values of r2 and r3 flow through the busses to the ALU. The ALU performs the right operation (it adds them together) and puts the result on bus2. This result is read by register r2.

When the clock now goes to 0, the sum of r2 and r3 is stored in r2 and PC is incremented.

2.3.4.3 A conditional jump (jpz)

The last instruction is 'jpz r15'. This instructs the processor to jump to another location in the code RAM (given by the value of r15), but only if the 'zero flag' is set (meaning that the result of the last calculation was zero). The instruction looks like: `1111 0000 0010 1111`.

The control logic will notice that the instruction begins with 1111. It will therefore inspect bits 4-6 from the instruction. As these bits are equal to 010, the status of the zero flag is inspected. If the zero flag is equal to 1, the following signals are sent:

- register r15 is asked to put its value on bus1
- register PC is asked to read its value from bus1
- the ALU is disconnected
- bus0 and bus2 are not connected to any registers

However, when the zero flag was equal to 0, the following signals are sent instead:

- register PC is asked to increment
- the ALU is disconnected
- bus0, bus1, and bus2 are not connected to any registers

When the clock goes to 0, register PC will contain the right value, depending on the state of the zero flag.

Chapter 3

The Moncky-2 processor

3.1 Instruction set

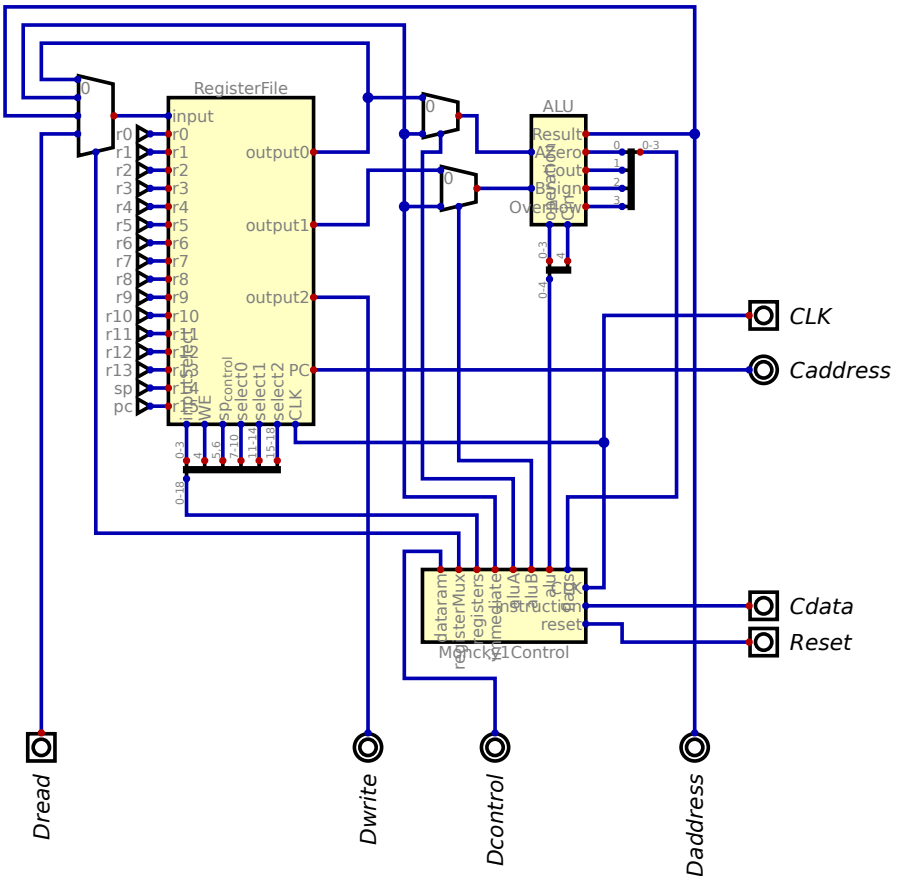
The Moncky-2 processor has exactly the same functionality as the Moncky-1. It only adds two instructions:

- reset (opcode 0010): will put the program counter back to 0. The processor automatically executes this instruction when the external 'Reset' input is set to 0.
- pcto (opcode 1110): will save the program counter to a register. This instruction was originally also available in the Moncky-1 but was removed to make the instruction set as small as possible. This instruction makes procedure calls easier so it was reintroduced.

3.2 Architecture

The architecture of the Moncky-2 processor is very different. The design can be implemented on an FPGA as there are no busses any more. They are replaced by MUXs.

The architecture is shown in the following diagram:



The registers have one input and three outputs. This is much easier to implement than the Moncky-1 registers. The control inputs determine which register is put on which output and which register is connected to the input. The input can come from 4 sources: another register (from output0), an immediate (from the control logic), the ALU, or the data RAM.

Register r15 is used as program counter (PC). It will increase automatically at the falling edge of the clock except when a new value is loaded in it. The jump instruction can therefore also be implemented as a 'nop r15, r...' instruction.

Both the inputs of the ALU can be connected to a register or an immediate. This is not necessary for the Moncky-2 instruction set but it enables other instructions, introduced in the Moncky-3.

The address sent to the data RAM always comes from the ALU. This is not necessary but again allows for future expansion of the instruction set (see Moncky-3).

Although the architecture is much different, the functionality remains the same. The Moncky-2 processor can still execute each instruction in one clock cycle.

Even though it is possible to make an FPGA version of this processor, this has not been done. Instead the architecture was used to create the Moncky-3 processor which has a lot more instructions. There is however a fully working implementation in Digital available:

`https://gitlab.com/big-bat/moncky`

Chapter 4

The Moncky-3 processor

The Moncky-3 processor is an extension of the Moncky-2 processor in many different ways:

- register r14 can be used as a stack pointer. It can increment or decrement while being used
- an interrupt line was added. When interrupts are enabled, the processor will respond to a rising edge on this input by calling a fixed address in memory. The stack is used to be able to return to the original code.
- there are many instructions (currently 32) that enable the code to be much faster. Every instruction still executes in one clock cycle.
- the Moncky-3 processor exists as a simulation in Digital. It has also been implemented on an FPGA using Verilog as an HDL. It forms the heart of the Moncky-3 computer which is explained in chapter 6.

4.1 Architecture

The architecture is based on that of the Moncky-2 processor. This means that:

- all connections are made using MUX's so it is possible to implement the processor on an FPGA
- the program counter is stored in register r15. This register automatically increments when it is not written to. If the register is read, the output is set to PC+1. In this way the return address can be read immediately when a procedure call is done.
- register r14 can be used as a stack pointer. There are two control lines indicating what should happen to this register:
  - 00 indicates that r14 is used as a normal register
  - 01 increments r14. This is used to 'pop' a value from the stack
  - 10 decrements r14. This is used to 'push' a value on the stack
  - 11 decrements r14. This is used for a hardware interrupt. When the control lines are both equal to 1, the value of r15 will be equal to PC (and not PC+1) so that the processor will return to the current instruction after the interrupt (instead of to the next).

There are also some differences:

- the processor has an 'enable' input. If set to 0, it will hold the processor from executing an instruction even when the clock ticks. This input can also be seen as a 'hold' input which is active low. The enable input is used to control the write enable inputs on all flip flops in the processor. This can for instance be used to synchronise the processor with the memory.
- the processor has input/output instructions. They behave like memory input/output instructions but they cause the 'io_request' output to be set to 1. This allows the use of separate memory for input/output.
- the processor contains interrupt control circuitry. Three flip flops are used to control the interrupts:
  - 'interrupts enabled' indicates if the interrupts are accepted or not. The state of this flip flop can be controlled by the instructions 'ei' (enable interrupts) and 'di' (disable interrupts).
  - 'last interrupt' is used to hold the previous value of the interrupt line. It is used to detect a rising edge.
  - 'interrupt granted' indicates if the requested interrupt is granted. It will only be equal to 1 during one clock cycle. This signal causes the processor to ignore the current instruction and perform a hardware interrupt instruction.

4.2 Instruction set architecture (ISA)

The ISA of the Moncky-3 consists of 4 groups of instructions:

- instructions without operands
- instructions with one operand of 4 bits
- instructions with two operands of 4 bits each
- instructions with three operands of 4 bits each or two operands of 4 and 8 bits respectively

The group is determined by the values of the least significant bits of the instruction as follows:

instruction	group	meaning
`pppp 0000 0000 0000`	0	opcode=pppp, no params
`rrrr pppp p000 0000`	1	opcode=ppppp, 1 param rrrr
`rrrr ssss pppp 0000`	2	opcode=pppp, 2 params rrrr and ssss
`rrrr ssss tttt pppp`	3	opcode=pppp, 3 params rrrr, ssss, and tttt
`rrrr iiii iiii pppp`	3	opcode=pppp, 2 params rrrr and iiiiiiii

Remark that the opcode of group 1 has 5 bits, which overlaps with group 2. This means that opcodes 0000 and 1000 of group 2 are not valid (they belong to group 1).

The same happens for opcode 00000 in group 1. This overlaps with group 0 and is therefore not allowed.

The following instructions are available:

machine language	meaning	example
0000 0000 0000 0000	PC=0 and disable interrupts	reset
0001 0000 0000 0000	PC stops incrementing, interrupts are enabled	halt
0010 0000 0000 0000	enable interrupts	ei
0011 0000 0000 0000	disable interrupts	di
0100 0000 0000 0000	return from interrupt	reti
0101 0000 0000 0000	software interrupt (call 0x10)	int
0110 0000 0000 0000	hardware interrupt (don't use this)	
rrrr 0000 1000 0000	RAM[sp--] = R[r]	push r3
rrrr 0001 0000 0000	R[r] = RAM[++sp]	pop r4
rrrr 0001 1000 0000	RAM[sp--] = PC; PC = R[r]	call [r5]
rrrr 0010 0000 0000	R[r] = flags (4 bits, zero extended)	sflags r1
rrrr 0010 1000 0000	flags = 4 LSB of R[r] (*)	rflags r2
rrrr 0011 0000 0000	PC = R[r]	jp [r3]
rrrr 10ff f000 0000	R[r] = -flags[f] (0000 or FFFF)	snz r4
rrrr 11ff f000 0000	if (flags[f]) PC = R[r]	jpnc r5
rrrr ssss 0001 0000	R[r] = IN[R[s]]	in r3, (r4)
rrrr ssss 0010 0000	OUT[R[s]]=R[r]	out r1, (r10)
rrrr iiii iiii 0001	R[r] = 0000 0000 iiii iiii	li r6, 42
rrrr iiii iiii 0010	R[r] = R[r] | iiii iiii 0000 0000	lih r7, 100
rrrr iiii iiii 0011	R[r] = R[r] + ssss ssss iiii iiii (sign extended) (*)	addi r8, -42
rrrr iiii iiii 0100	R[r] = R[r] & 0000 0000 iiii iiii (*)	andi r9, 0x0F
rrrr iiii iiii 0101	R[r] = R[r] | 0000 0000 iiii iiii (*)	ori r10, 0x80
rrrr iiii iiii 0110	set flags for (R[r] - ssss ssss iiii iiii) (*)	cmpi r1, 32
rrrr iiii iiii 0111	set flags for (ssss ssss iiii iiii - R[r]) (*)	cmpir r2, 32
rrrr ssss AAAA 1000	R[r] = R[r] ALU R[s] (*)	add r4, r5
rrrr ssss AAAA 1001	set flags for (R[r] ALU R[s]) (*)	addf r5, r6
rrrr iiii AAAA 1010	R[r] = R[r] ALU 0000 0000 0000 iiii (*)	shli r7, 3
rrrr iiii AAAA 1011	set flags for (R[r] = R[r] ALU 0000 0000 0000 iiii) (*)	shrif r8, 2
rrrr ssss tttt 1100	R[r] = RAM[R[s] + R[t]]	lda r3, (r4+r1)
rrrr ssss tttt 1101	RAM[R[s] + R[t]] = R[r]	sta r1, (r4+r0)
rrrr iiii tttt 1110	R[r] = RAM[tttt + 0000 0000 0000 iiii]	ldi r3, (r4+7)
rrrr iiii tttt 1111	RAM[R[t] + 0000 0000 0000 iiii] = R[r]	sti r2, (r0+1)

All other possible instructions are currently wired to 'nop' (no operation).

There are 4 flags which are only set when an arithmetic operation is done (these instructions are marked with a (*) in the table above):

- zero flag: indicates if the result was equal to zero
- carry flag: for additions and subtractions it indicates if an extra carry-bit was generated. This indicates an overflow for unsigned numbers (for subtractions the inverse of the carry flag indicates an overflow!). When doing shift operations with offset 1, the carry flag indicates the value of the bit that was shifted out. If the offset is not equal to 1, the resulting carry flag is meaningless.
- sign flag: indicates if the result was smaller than zero (it just gives the value of the most significant bit of the result). This is only relevant for 2-complement numbers.
- overflow flag: after an addition or subtraction, this flag indicates if a 2-complement overflow

has occurred. When doing logical operations, this flag indicates the parity of the result by xor-ing all the bits (i.e. if an even number of bits are 1, the flag is set to 0)

4.3 Assembly language

This section describes all assembly instructions more in detail:

reset this will reset the processor by setting the program counter to 0. It will also disable interrupts so they cannot interfere.

halt this instruction causes the processor to stop executing instructions. The PC will not increment any more. The interrupts are enabled so that it is possible to escape from this state with a hardware interrupt

ei enables the interrupts

di disables the interrupts

reti returns from an interrupt. It is the same as 'pop r15' but this instruction will also enable the interrupts so they don't need to be explicitly enabled.

int causes the processor to perform a software interrupt. It is equivalent to 'call 0x10' (the interrupt handler is hard wired to address 0x10), but it will also disable interrupts automatically

hi hardware interrupt: this is not to be used by a programmer. It behaves like the 'int' instruction but the return address will be set to the current address (since the current instruction will not be executed if a hardware interrupt occurs)

push pushes a register to the stack. This can be used to save the value of a register or to pass the value to a subroutine

pop pops a register from the stack. This can be used to restore a value of a register that was previously saved on the stack

call this will jump to the address stored in a register. At the same time, the return address is pushed onto the stack

sflags saves the flags to a register. The flags consist of 4 bits (from LSB to MSB): zero, carry, sign, overflow. This instruction can be used in an interrupt handler so that the state of the flags is preserved

rflags restores the flags from a register. The 4 LSB of the register are copied to the flags.

jp this will jump to the value of the given register. This instruction is equivalent to nop r15, ... but it will not change the status of the flags (the nop instruction is regarded as a calculation and will change the flags)

sz,snz,sc,snc,ss,sns,so,sno these instructions will save the status of a flag to a register. If the selected flag is set, the result will be 0xFFFF (0 otherwise). One can also save the inverse of the flag. For instance, 'sz' will save the status of the zero flag, while 'snz' will save its inverse. These instructions are very useful when conditions need to be combined (with and, or, not, ...). They can also be used to avoid conditional jumps (see chapter 7).

jpz,jpnz,jpc,jpnc,jps,jpns,jpo,jpno these instructions will jump to the address stored in a register, but only if the selected condition is true

in reads data from input/output memory at the address stored in a register. The data is stored in the given register. For instance, 'in r0, (r1)' will use r1 for the address and will store the result into r0.

out writes data to input/output memory. For instance 'out r0, (r1)' will store the value of r0 at the address given by r1.

li loads a value (an 'immediate') into a register. The value must be positive and can only be up to 8 bits long. It is zero-extended to fit into the register.

lih loads a value into the MSB of a register. The LSB of the register will not be changed. The value must be a positive 8-bit number.

addi this will add a value to a register. The value must be an 8-bit 2-complement number. It is sign-extended before being added.

andi this will do a bitwise AND of a register and a value. The value must be an unsigned 8-bit value. It is zero-extended.

ori this will do a bitwise OR of a register and a value. The value must be an unsigned 8-bit value. It is zero-extended.

cmpi compares a register with a value by subtracting the value from the register. The result is not stored. Only the flags are stored. The value must be an 8-bit 2-complement value. It is sign-extended before being subtracted from the register

cmpir compares a register with a value by subtracting the register from the value. The result is not stored. Only the flags are stored. The value must be an 8-bit 2-complement value. It is sign-extended before subtracting the register.

nop,or,and,xor,add,sub,shl,shr,ashr,not,neg,addc,subc,shlc,shrc these are instructions that do an ALU operation on two registers. The flags are set and the result is stored into the first operand

nopf,orf,andf,xorf,addf,subf,shlf,shrf,ashrf,notf,negf,addcf,subcf,shlcf,shrcf these are instructions that do an ALU operation on two registers. The result is not stored. Only the flags are set

nopi,xori,subi,shli,shri,ashri,noti,negi,addci,subci,shlci,shrci these instructions do an ALU operation on a register and an immediate value. The value must be an unsigned 4-bit number. The result is stored in the given register. The addi, ori, and andi instructions are left out because there exists a better version (described above) with 8-bit values. However, the opcodes for these instructions exist and could be used.

nopif,xorif,subif,shlif,shrif,ashrif,notif,negif,addcif,subcif,shlcif,shrcif these instructions do an ALU operation on a register and an immediate value. The value must be an unsigned 4-bit number. The result is not stored. Only the flags are set.

lda will load a value from the data memory. The address is given by the sum of the values of two registers and the result is stored in another register. This instruction can be useful to access elements in an array.

sta will store a value to data memory. The address is given by the sum of the values of two registers.

ldi will load a value from the data memory. The address is given by the sum of the value of a register and an immediate value. The value must be an unsigned 4-bit number. If the immediate is equal to 0, this corresponds to the 'ld' instruction of the Moncky-1.

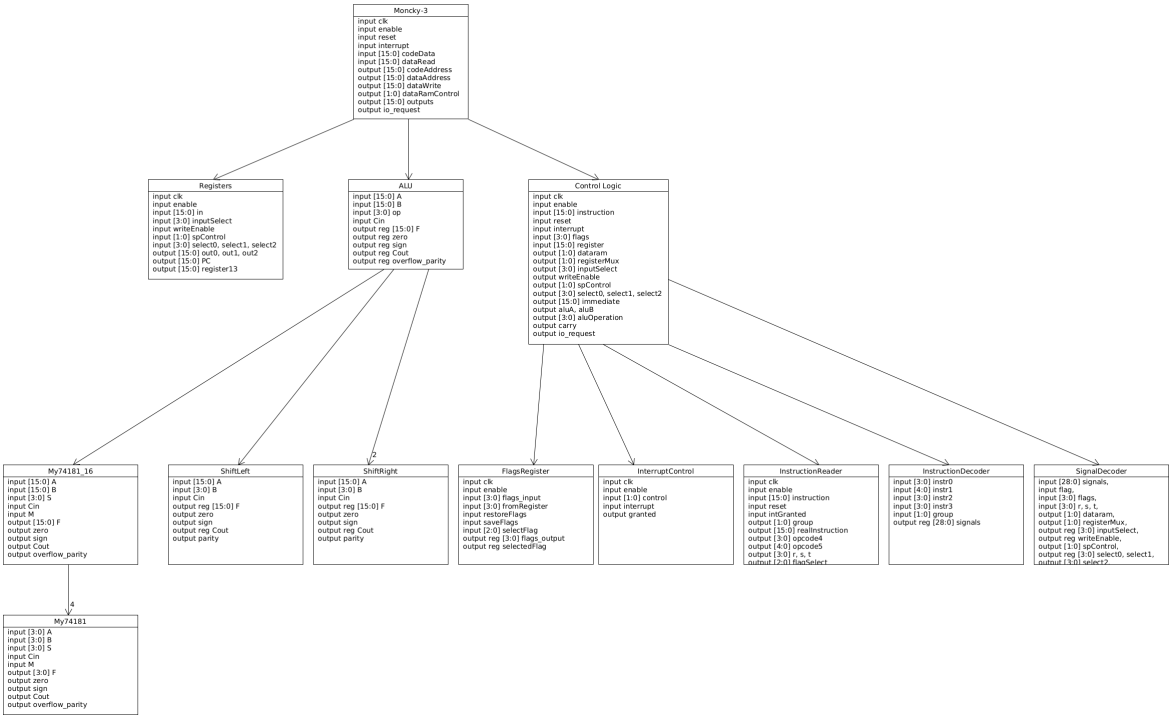
sti will store a value to data memory. The address is given by the sum of the value of a register and an immediate value. The value must be an unsigned 4-bit number. If the immediate is equal to 0, this corresponds to the 'st' instruction of the Moncky-1.

4.4 Implementation in Verilog

The Moncky-3 processor was implemented in Verilog HDL. You can find the code on gitlab:

`https://gitlab.com/big-bat/moncky`

A schematic overview of all the modules can be seen here:



The Moncky-3 processor has the following connections:

- clk: this is the main clock. Each instruction is performed at the falling edge of the clock
- enable: if 0 the clock is disabled. This can be used to synchronise the processor with memory
- reset: active high. If 0 the processor performs the 'reset' instruction
- interrupt: the processor will perform a hardware interrupt when a rising edge on this connection is detected and interrupts are enabled
- codeAddress, codeData: these are used to read instructions from the code memory
- dataAddress, dataRead, dataWrite, dataRamControl: these are used to read and write data from and to the data memory
- outputs: this is connected to register r13. It allows to directly drive external hardware using this register

- io_request: this indicates an input/output request. Used in conjunction with codeAddress, codeData, dataWrite, and dataRamControl

The Moncky processor contains an ALU which is based on the 74181 chip. The circuit was altered so that it uses positive logic for data and control signals. Four of these circuits are then combined into a 16-bit version. The shift operations are implemented separately. MUX's are used to select the right result (74181 or shift circuits).

The control logic consists of different parts:

- FlagsRegister is a 4-bit register containing the values of the flags
- InterruptControl contains the 3 flip flops to control the interrupts
- InstructionReader disassembles the instruction into its different parts
- InstructionDecoder decodes the instruction into control signals that are internally used (in the control logic)
- SignalDecoder will translate the control signals into control signals for the ALU, the registers, and the MUX's

Chapter 5

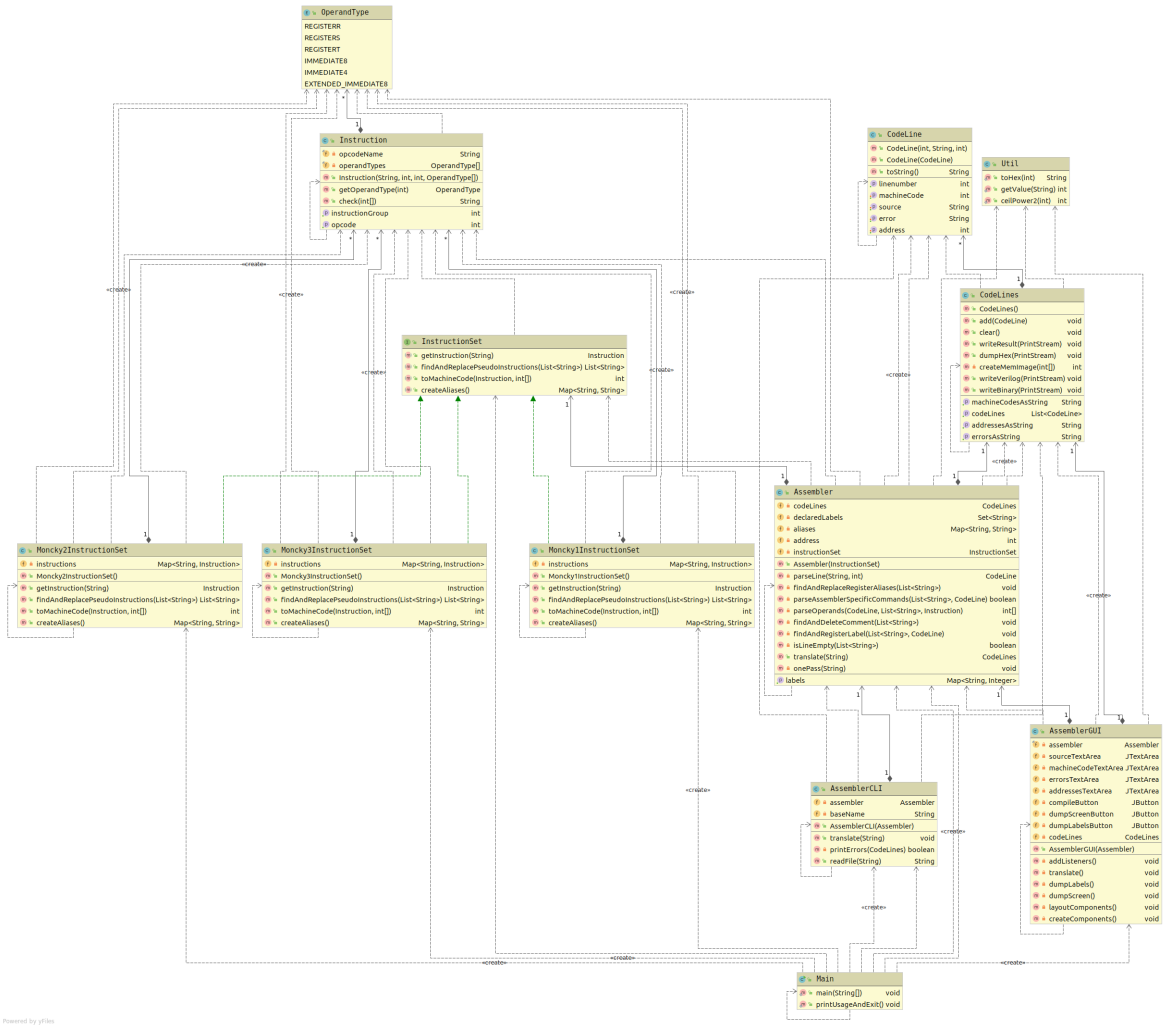
The Moncky assembler

The Moncky assembler is a Java program that can translate Moncky assembly language to machine code. It currently supports all three Moncky processors.

The coed can be found at:

`https://gitlab.com/big-bat/moncky`

The code was made so that extension to other processors can be done very easily. The class diagram looks like this:



The program starts in the Main class. There must be at least one command-line parameter which determines the instruction set used. It must be one of 'Moncky1', 'Moncky2', 'Moncky3'. If only one parameter is given the AssemblerGUI is used to show a simple graphical user interface in which the user can type code and see the resulting machine code directly. When two or three parameters are given, the AssemberCLI is used. This has no graphical interface. It has the following extra command-line parameters: an optional path to the firmware files and the source file (must end in '.asm'). If a path to firmware is given, the files are added to the source.

Parsing is done in a very simple way. Every line of the source is regarded as one instruction (or comment). There are no multi-line instructions possible nor is it possible to put more than one instruction on one line. Each line is split into its tokens (called 'parts'). It uses spaces, tabs, braces, square braces, and the plus sign as white space. Characters can be specified between single quotes and string can be given with double quotes. Strings are automatically null-terminated and encoded with 8 bits per character (2 characters per word).

All instructions consist of an opcode followed by operands.

The parser is a two-pass parser. In the first stage, all code is read and labels are set to the correct values. In the second stage code is generated with the right values for all labels.

The output of the assembler consists of 4 files:

- a '.txt' file that contains the source code together with the generated code in a readable form
- a '.hex' file that can be used to load a ROM in Digital or Logisim
- a '.v' file that contains Verilog code that describes a ROM with the assembled code. This file can be used in the Verilog implementation of the Moncky-3 processor
- a '.img' file containing a binary version of the code. This can be used to store in a file on an SD card. The Moncky-3 computer can read this file into memory and execute it

5.1 Comments

Comments can be put in the code using ';'. The comment extends to the end of the line. This is comparable to '//' in the C programming language or '#' in scripting languages.

5.2 Assembler specific instructions

The assembler adds additional instructions that all start with a period '.':

.org with an address as parameter will set the current address to the given value

.def with a label and a value defines a label with the given value. The label must start with ':'. This can for instance be used to define the location of a variable or the value of a constant

.data followed by literals can be used to fill the memory with numbers. The literals can be decimal, hexadecimal, octal, or binary numbers. They can also be defined as a character if put between single quotes, or as a single string between double quotes (the string will be encoded with 8 bits/char and will be null-terminated).

.alias defines an alias. It is similar to the .def instruction, but aliases can also be 'unaliased'. An alias must start with a '$'. Aliases can be used to replace register names with variable names when a register is used to contain its value.

.unalias removes an alias

.rmAliases removes all aliases

5.3 Labels

In the beginning of each line a label can be specified. This will associate the label with the current address. Labels always start with a ':' character.

They can be referenced in an operand using ':' or '::'. The following code demonstrates this:

```
            andi r0, 255
            li r0, :addr
            lih r0, ::addr
            jpz [r0]
            ...
    :addr   li r0, 42
            sub r0, r1
```

This code will jump to ':addr' when the value of the LSB of r0 is equal to zero.

The double colon '::' is used to get the MSB of the label.

As shown a label does not need to be defined before it is used.

5.4 Virtual opcodes

The assembler can add other opcodes that make code more readable. They map on other existing opcodes. For the Moncky-3 processor the following virtual opcodes are defined:

nop without any other operands will translate to 'nop r0, r0'. This causes the processor to do nothing but use one clock cycle (although it will set the flags according to the value of r0). It can be used for timing.

jpi with one immediate operand (addr) will translate to 'li r15, addr'. It will cause the processor to jump to a specific address. The address must fit in an 8-bit unsigned number. This is handy when writing small programs

jpr with one operand (a register r) will translate to 'add r15, r'. This will jump to an address relative to the current address. This can be used to write code that needs to be relocated

jpfi with one immediate operand (offset) will translate to 'addi r15, offset'. This will jump forward to an immediate offset relative to the current address.

jpbi with one immediate operand (offset) will translate to 'subi r15, offset'. This will jump backward to an immediate offset relative to the current address.

set with two operands (registers r and s) will translate to 'nop r, s'. This copies the value of register s to register r.

st with two operands (registers r and s) will translate to 'sti r, (s+0)'. It is equivalent to the 'st' instruction on the Moncky-1 processor.

ld with two operands (registers r and s) will translate to 'ldi r, (s+0)'. It is equivalent to the 'ld' instruction on the Moncky-1 processor.

ret with no operands translates to 'pop pc'. It returns from a subroutine by taking the return address from stack.

cmp with two operands (registers r and s) will translate to 'subf r, s'. It will cause the processor to subtract the two registers, only setting flags (not storing the result)

inc with one operand (register r) will translate to 'addi r, 1'. It increments a register

dec with one operand (register r) will translate to 'subi r, 1'. It decrements a register

5.5 Aliases

As described before, aliases can be defined using the .alias instruction. The Moncky-3 instruction set defines the following aliases by default:

- pc: this is an alias for r15 which is the program counter
- sp: this is an alias for r14 which is used as a stack pointer
- bp: this is an alias for r12 which can be used as a base pointer

5.6 Operands

Operands can be a register, a label, or a number.

A register must always begin with the character 'r' followed by a number between 0 and 15. It can also be an alias defined by the instruction set (sp, bp, pc).

Labels always start with a ':' so they can easily be recognised. Normally a label is replaced by the LSB of the value of the label. If the label starts with '::' the MSB is used. This can be used to quickly load an address into a register. For instance:

```
li r0, :addr
lih r0, ::addr
jp [r0]
```

This will load the value of :addr into r0 and jump to that location.

Numbers can be specified in decimal, hexadecimal (starting with 0x), octal (starting with 0), binary (starting with 0b), or as a character (between single quotes). They are checked to have the correct number of bits.

Chapter 6

The Moncky-3 computer

The Moncky-3 computer is a retro computer built around the Moncky-3 processor. The computer features 128 KB of RAM, has a VGA output, a PS/2 keyboard input and uses SD cards as 'floppies'. It can be used to play games or program in a simple interpreted programming language.

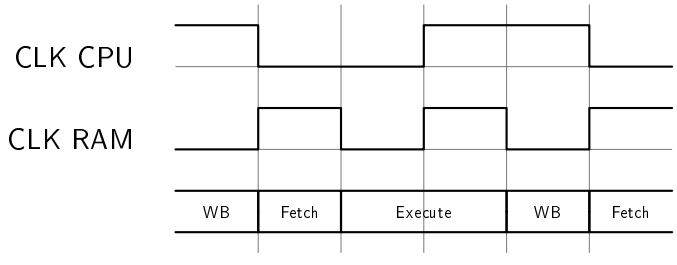
6.1 The RAM

In order to make the computer flexible, it should be possible to load a program into memory (as data) and execute it. The Harvard architecture does not allow that since data is stored in a different location.

Therefore the Harvard architecture is altered in the Moncky-3 computer so that code and data can be in the same RAM. This is done using dual-port RAM. In this way the processor does not need to be aware that code and data are stored in the same location. It can however load code into memory and then jump to it. The dual-port RAM has one port for reading and writing data and another for reading the code.

The dual-port RAM is made with BRAM on the FPGA. There is however a big drawback doing this: BRAM is synchronous memory. This means it will only read and write data at the rising edge of the clock. For writing this is a good thing because the address can stabilise before the data is written to a specific location. But for reading it causes an extra delay. For instance, the Moncky-3 processor will put the address of the next instruction at a falling edge of the clock. It has to wait for the next rising edge to be able to read the instruction at that address.

In order to overcome this problem, there are two possible solutions. The first is to double the clock speed of the RAM and let it respond to the falling edge of its clock. This results in the following:

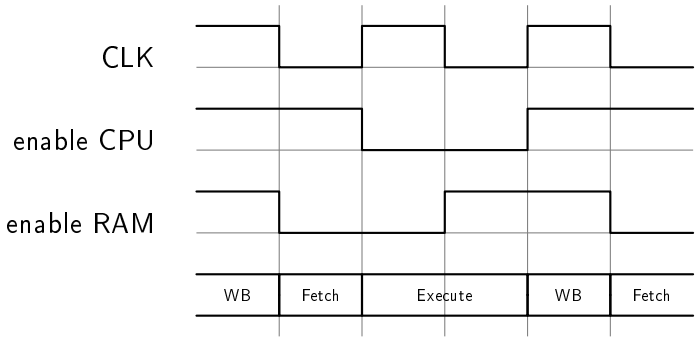


There are now 3 distinct phases: fetch, execute, and write back (WB). In the first phase the processor will put the value of PC on the address bus. The memory will output the instruction

at the end of the fetch cycle. Now the processor can execute the instruction. If the processor needs to read data from the RAM, the data will become available in the WB phase. If the processor needs to write data to the RAM, this is also done during the WB phase. Remark that these phases have a high correlation with the Von Neumann cycle. There is however no distinct decode phase.

Doubling the clock of the memory (or decreasing the cpu clock speed) works very well, but on an FPGA this poses a problem for the development tools. There are now two 'clock domains' that are mixed together. The development tools try to check all the timing constraints. But this cannot be done if a mixture of clock domains is present, unless there is a buffer in between them (this problem is also known as 'Clock-Domain Crossing'). Therefore another approach is used.

In this approach both the CPU and the RAM get an 'enable' signal. Their clock speed is the same, but the enable signals will enable the clock pulse when needed. This is depicted in the following timing diagram:



The RAM is now configured to respond to the rising edge of the clock. But data will only be written when the enable RAM signal is high. This creates the same effect as before, but now there is only one clock domain which makes it easier to check all the timing constraints.

6.2 Clock domains

As discussed before having different clock domains can solve problems, but they can also create them. The RAM and CPU are in the same clock domain, but other components need other clock speeds. The following clock domains are present in the Moncky-3 computer:

- CLK100MHZ: this is the base clock of the FPGA. It is used to derive the other clocks
- clk_cpu: this is the clock given to the cpu. It currently runs at 20 MHz. Since the enable signal is only active once in two clock cycles the cpu effectively runs at 10 MHz.
- clk_mem: this clock is the same as clk_cpu.
- clk_video: this clock is used to generate the VGA signal. For a 640x480 resolution a frequency of 25 MHz is needed.
- clk_io: this clock is given to the keyboard controller, the counters, and the SPI controller. Its frequency is 12.5 MHz.

The FPGA contains a clock generator that can derive the clocks. It outputs a 'locked' signal that becomes high when all clocks are in sync. This signal is used to drive the reset pin of all components so they all reset as long as the clocks have not been synchronised yet.

6.3 The ROM

Having one memory for data and code is very useful, but when the computer boots up, some code must already be available so it can start up. To enable this a ROM is added to the system that will 'shadow' the RAM. When the processor reads code from the lower part of the memory (addresses 0x0000 till 0x0FFF), the data doesn't come from RAM, but from the ROM. There is a MUX in the system that will select the ROM when the 4 most significant bits of the address are equal to 0.

When the processor writes data to the RAM it will be stored in the RAM. Even if the address is low the data still goes to RAM. This is used in the boot sequence. The ROM shadowing can be turned off. If there is code in the RAM that code will take over.

The ROM contains the following code:

- clear the screen
- plot a dot on the screen
- draw a character on the screen
- print a string
- print a hexadecimal number
- scroll the screen up
- wait for a key press
- detect and initialise an SD card
- read a block from the SD card
- read a file from a FAT16 file system on an SD card

When the computer boots the system will show a splash screen and start looking for an SD card. If no SD card is present the computer shows an error and reboots after a key press. If an SD card is detected, the file 'MONCKYOS' is expected to be on the first partition on a FAT16 file system. Currently the partition should be smaller than 32MB. The file is loaded at address 0x0000 in memory. After this the ROM shadowing is turned off and the computer jumps back to address 0x0000.

The ROM shadowing can be turned off by writing the number 0xAAAA in the location 0xFFFF. This means that this location cannot be used for any other purpose.

6.4 Input/output

In order to have as much memory available as possible the input/output is not memory mapped. In stead the io_request pin together with the opcodes 'in' and 'out' are used to access the input/output. This creates a new address space of 65536 entries. A specific i/o component in the computer regulates the data flow between the i/o address space and all the components.

Next to input/output the i/o address space is also used by the firmware to store data. In the next list, they are marked with a '*'.

The following addresses are currently used (more information about the exact values is given per i/o component) (remark that the order of the values is a bit chaotic, due to the way the software and hardware evolved):

- 0x0000 - 0x7CFF: video screen buffer 0
- 0x7D00: video configuration
- 0x7D01: keyboard last key press
- * 0x7D02: cursor position (LSB=x, MSB=y)
- * 0x7D03: shift pressed
- * 0x7D04: background colour
- * 0x7D05 - 0x7D25 : saved image behind cursor
- * 0x7D26: graphics draw colour
- * 0X7D27: graphics draw buffer
- 0x7E00: SPI data
- 0x7E01: SPI control
- 0x7E02: LSB of usec
- 0x7E03: MSB of usec
- 0x7E04: LSB of msec
- 0x7E05: MSB of msec
- * 0x7E06: sd_type (1,2, or 3 for SD1, SD2, and SDHC resp.), high byte contains error if it occurred
- * 0x7E07 - 0x7E0B: sd_result (5 words containing the result to a command)
- 0x8000 - 0xFCFF : video screen buffer 1
- * 0xFD00 - 0xFDFF: fat_sector
- * 0xFE00 - 0xFE01: fat_startPartition
- * 0xFE02 - 0xFE03: fat_lengthPartition
- * 0xFE04: fat_sectorsPerFat
- * 0xFE05: fat_fatPosition
- * 0xFE06: fat_rootFolderPosition
- * 0xFE07 - 0xFE08: fat_rootFolderPosition
- * 0xFE09: fat_fileStart
- * 0xFE0A - 0xFE0B: fat_fileLength
- * 0xFE0C: fat_currentClusterNumber
- * 0xFE0D: fat_sectorsPerCluster
- * 0xFE0E: fat_currentSectorNumber

6.5 The VGA controller

The VGA controller generates the signals to drive a VGA monitor at a resolution of 640x480 at 60 Hz. Due to the limitation of the video memory the pixels are grouped in 2x2 pixels. Only the middle 400 lines of the screen are used. This means that the effective resolution is 320x200. When displayed on a 4:3 monitor the pixels are square.

The VGA controller only has a graphics mode. There is no text mode. That means that text is rendered by drawing pixels. While this is much less efficient when only text is needed, it is much more simple and flexible. Text and graphics can for instance easily be mixed.

The values for the colour of each pixel are stored in a separate RAM (called VRAM). Eight bits are used per pixel as follows:

red	red	red	green	green	green	blue	blue

Since the memory contains 16 bit numbers, each memory cell holds the values of two pixels. The LSB defines the left pixel and the MSB the right pixel. This means that the video memory is at least 320x200 bytes long. This equals to 32000 words of 16 bit.

The total video memory however is 65536 words long. This means that 2 screen buffers can fit in the VRAM. This makes it possible to do double buffering animations. The idea is that the computer will write to one buffer while the other is shown on the screen. After the buffer is ready the VGA controller is asked to switch the active buffer. In this way a fluent animation can be done without flickering. Switching buffers should be done at the vertical retrace. The memory location 0x7D00 is used to specify which buffer needs to be displayed (0 or 1). The processor can write the right value in this location and the VGA controller will immediately switch to the new buffer.

In order to detect the vertical retrace of the VGA controller, it will send a hardware interrupt to the processor. The processor thus gets 60 interrupts per second from the VGA controller. This interrupt might also be used to do other tasks (like multitasking).

The VRAM is organised as follows:

- 0x0000 - 0x7CFF: video buffer 0
- 0x7D00: video control register (defines which buffer needs to be displayed)
- 0x7D01 - 0x7FFF: free for other uses (this address space is used for other I/O components and firmware variables)
- 0x8000 - 0xFCFF: video buffer 1
- 0xFD00 - 0xFFFF: available for other uses

6.6 The keyboard controller

The keyboard controller is able to interface with a PS/2 keyboard. It is only capable of receiving data from the keyboard. This means that the LEDs on the keyboard do not function. When a key is pressed the keyboard will send the scan code of the key to the keyboard controller. The data is sent one byte at a time as follows: 1 start bit, 8 data bits, a parity bit, and a stop bit. Most scan codes are 8 bits long but there are also keys with a 2-byte scan code. When a key is released the keyboard will first send a 'break' code followed by the scan code of the key.

The keyboard controller is split into 3 parts: the keyboard debouncer, keyboard interface, decoder, and buffer.

6.6.1 The keyboard debouncer

A PS/2 keyboard generates signals for the key pressed as well as a clock. This clock runs at about 10KHz. Since the wires between the keyboard and the computer are quite long and since the computer runs at a much higher speed the signals can be distorted. Therefore they must be debounced. When the value of the input doesn't change for 2.56 usec, the value is passed on to the keyboard interface.

There is a debouncer for both the data and the clock input.

6.6.2 The keyboard interface

The keyboard interface receives bytes of data from the keyboard. It does this by waiting until 11 bits are received. The start bit, stop bit, and parity is checked. When a byte was successfully received it is presented at the output and a 'ready' signal is put high.

The keyboard interface is in fact a finite state machine having 2 states: 'waiting' and 'receiving'. There is a watchdog timer that will force the machine into 'waiting' mode when the machine stays too long in 'receiving' mode.

If an error occurs the keyboard interface will set its 'error' output to high.

6.6.3 The keyboard decoder

The keyboard decoder will receive the bytes of the keyboard interface and turn them into one 16-bit number. Bits 8-0 contain the scan code of the key that was pressed or released. Bit 9 will indicate if the key was pressed or released.

The keyboard decoder is implemented as a finite state machine. It has 4 states: 'waiting', 'extended', 'released', 'extended released'. In the first state the decoder is waiting for an incoming byte. If this byte indicates an extended scan code the state turns into 'extended'. If a break code is received the state changes to 'released' or 'extended released' according to the current state.

When a complete scan code has been detected the value is passed to the keyboard buffer.

6.6.4 The keyboard buffer

The keyboard buffer contains a circular FIFO queue. It can hold up to 15 key strokes. The keyboard buffer will store all incoming key strokes from the keyboard decoder. When the processor reads from keyboard register (mapped at address 0x7D01) the keyboard buffer will deliver the data and remove it from the buffer.

This component takes away the need for an interrupt whenever a key is pressed. A drawback is that the keyboard buffer will not be emptied automatically when a reset occurs. This needs to be done in software by reading the buffer until it outputs a value of 0.

6.7 The counters

In order to measure time 2 counters run in hardware and are available to the processor. The first counter counts microseconds since startup and the other count milliseconds since startup. Both counters are 32 bits long and are stored in the following locations in i/o memory:

- 0x7E02: LSB of usec

- 0x7E03: MSB of usec
- 0x7E04: LSB of msec
- 0x7E05: MSB of msec

The counters can be used by the processor when it needs to wait for a specific amount of time, or to detect a timeout.

A possible extension of this component would allow it to send an interrupt to the processor in a configurable amount of time. This is currently not implemented.

6.8 The SPI controller

The SPI controller can control up to 4 SPI slaves. The configuration register (located at 0x7E01) contains the following data:

d7	d6	d5	d4	d3	d2	d1	d0	ss3	ss2	ss1	ss0	ready	m1	m0	transfer

The values d7 - d0 define the speed of the SPI clock. If one wants to set the clock to a certain frequency, the value must be set according to the following formula:

$$d = \frac{12,500,000}{2 \cdot f} - 1$$

The values ss3 - ss0 are the slave-select signals for each of the slaves. At most one of these bits should be 0 at any time.

The values m1 - m0 define the SPI mode (CPHA and CPOL).

The ready bit will be put to 1 when the SPI controller is waiting to send and receive data.

When the processor puts the value of 'transfer' from 0 to 1 (rising edge), the SPI controller will put the 'ready' bit to 0 and start sending the data that is present in location 0x7E00. The data in this location is automatically overwritten by the received data. After sending and receiving the 'ready' bit is put back to 1. The transfer bit will always be read as 0.

6.9 Statistics

The Moncky-3 computer is implemented on an Xilinx Spartan 7 FPGA. It uses the following resources:

```
Report Cell Usage:
+------+---------+------+
|      |Cell     |Count |
+------+---------+------+
|1     |clk_wiz  |     1|
|2     |CARRY4   |    39|
|3     |LUT1     |    77|
|4     |LUT2     |   127|
|5     |LUT3     |    92|
|6     |LUT4     |   174|
```

```
|7     |LUT5     |   171|
|8     |LUT6     |   735|
|9     |MUXF7    |    73|
|10    |MUXF8    |    19|
|11    |RAM32M   |     2|
|12    |RAMB36E1 |    66|
|18    |FDRE     |   593|
|19    |FDSE     |    16|
|20    |IBUF     |     4|
|21    |OBUF     |    19|
+------+---------+------+
```

It consumes 0.251 W of power.

Chapter 7

Coding tricks

This chapter contains some coding tricks and hints that can be used when programming the Moncky-3 processor.

7.1 Loading 16 bit numbers

The 'li' instruction loads an 8 bit immediate number into a register. But in many cases one needs to load 16 bits into a register.

For this reason the 'lih' instruction was added to the instruction set. It loads an 8-bit immediate into the MSB of a register, leaving the LSB intact.

As an example loading the number 0x1234 into register r5 can be done as follows:

```
li r5, 0x34
lih r5, 0x12
```

Remark that switching the instructions will not work since the 'li' instruction clears the MSB of the register.

In order to load the value of a label into a register, one can use the ':' and '::' notation:

```
li r5, :address
lih r5, ::address
```

The double colon notation denotes the MSB of the label.

7.2 32 bit arithmetic

In order to do 32 bit arithmetic the carry bit can be used. Two examples are given here. It is assumed that registers r0 and r1 contain a 32 bit number. The LSB are in r0 and the MSB are in r1. Another number is present in registers r2 (LSB) and r3 (MSB).

Adding these numbers can be done as follows:

```
add r0, r2
addc r1, r3
```

The result is now in r0 and r1.

Subtracting these numbers can be done in a similar way:

```
sub r0, r2
subc r1, r3
```

7.3 Calling convention

There are many ways to call a subroutine with parameters. In the Moncky-3 firmware the following convention is always used (register r12 is called 'bp' which stands for 'base pointer'):

- parameters are pushed on the stack in the order of appearance (i.e. the parameters are in reversed order in memory). If a parameter is longer than 16 bits the MSB are pushed first.
- the subroutine can then be called (pushing the return address on stack)
- if there are parameters or local variables the subroutine will do 'push bp' and 'set bp, sp' to set the base pointer to the current stack frame
- the subroutine will then make room for local variables on the stack (with 'subi sp, ...')
- the subroutine can access the parameters with: 'ldi r0, (bp+3)'. bp+3 will always be the last parameter
- the subroutine can access the local vars using 'lda r0, (bp+r1)' where r1 holds the negative offset of the variable. bp-0 is the first local variable. If no other values were pushed onto the stack, the stack pointer can also be used with an immediate: 'ldi r0, (sp+1)'. sp+1 will be the last local variable.
- the subroutine is not expected to save the value of any register other than sp and bp.
- the subroutine puts the return value in r0. If more than 16 bits need to be returned register r1 is used to hold the most significant bits of the value
- the subroutine ends by freeing up the space for local variables ('addi sp, ...'). Then it does 'pop bp' and finally 'ret' to return to the caller
- after this the parameters are removed from stack with 'addi sp, ...'

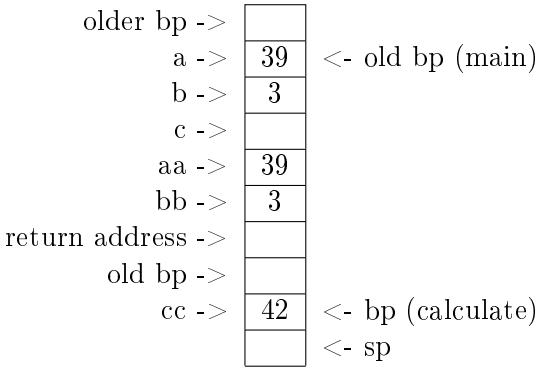
Here is an example in C:

```
int calculate(int aa, int bb) {
  int cc;
  cc = aa + bb;
  return cc;
}
void main() {
  int a = 39;
  int b = 3;
  int c;
  c = calculate(a, b);
}
```

This can be translated (without any optimisation) as follows:

```
:calculate
push bp
set bp, sp
subi sp, 1      ; make room for cc
ldi r0, (bp+4)  ; read value of aa
ldi r1, (bp+3)  ; read value of bb
add r0, r1      ; add aa and bb
sti r0, (sp+1)  ; store cc
ldi r0, (sp+1)  ; set return value to cc
addi sp, 1      ; clean up cc
pop bp
ret
:main
push bp
set bp, sp
subi sp, 3      ; make room for a, b, and c
li r0, 39
sti r0, (sp+3)  ; store 39 in a using sp
li r0, 3
sti r0, (sp+2)  ; store 3 in b using sp
ldi r0, (bp+0)  ; load value of a using bp
push r0         ; first parameter aa
li r0, 1
neg r0, r0
lda r0, (bp+r0) ; load value of b using bp
push r0         ; second parameter bb
li r0, :calculate
lih r0, ::calculate
call [r0]
addi sp, 2      ; clean up parameters
li r1, 2
neg r1, r1
st r0, (bp+r1)  ; store return value in c
addi sp, 3      ; clean up local vars a, b, and c
pop bp
ret
```

The stack frame will look as follows (while executing calculate()):



7.4 Using masks

Masks can be used to set, unset or toggle individual bits without altering the other bits.

To set bits in a register, create a mask that has a 1 for each bit that needs to be set. Then 'or' the mask with the register. For instance, setting bits 0 and 5 in register r0 can be done as follows:

```
li r1, 0b00100001 ; mask
or r0, r1
```

To unset bits in a register, create a mask that has a 0 for each bit that needs to be unset. Then 'and' the mask with the register. For example, resetting bits 0 and 5 in register r0 can be done as follows:

```
li r1, 0b00100001 ; mask
not r1, r1        ; take inverse
and r0, r1
```

Bits can also be toggled using the 'xor' operation. For instance, toggling bits 0 and 5 in register r0 can be done as follows:

```
li r1, 0b00100001 ; mask
xor r0, r1
```

7.5 Unsigned overflow

The carry flag can be used to detect overflow for unsigned numbers. If two unsigned numbers are added, the carry flag will be 1 if an overflow occurred.

But this is not true for subtractions. In this case the inverse carry flag will indicate the overflow. This is because a subtraction is executed as an addition using 2-complement.

In fact $a - b$ is implemented as $a + \overline{b} + 1$.

The carry flag is not the same as a 'borrow' flag. The processor will set the carry flag to the value of the 'extra bit' that can appear. Some processors will invert this bit for a subtraction but the Moncky processor doesn't do that.

The programmer must be aware if a subtraction or an addition was done and use the carry bit accordingly.

7.6 Comparing numbers

Conditional jumps use the flags to determine if a jump needs to be done. The flags are set when an arithmetic operation is performed. They are left unchanged when no arithmetic operation is executed.

The standard way to compare numbers is to subtract one from the other so that the flags are set. This section gives a quick overview on how to perform most comparisons.

Imagine that the r0 contains the value of variable 'a' and r1 contains the value of variable 'b'.

When a and b are 2-complement signed integers the following comparisons can be done:

a==b subf r0, r1; jpz ...

a!=b subf r0, r1; jpnz ...

a<b subf r0, r1; jps ...

a>b subf r1, r0; jps ...

a<=b subf r1, r0; jpns ...

a>=b subf r0, r1; jpns ...

If a and b are unsigned integers, the comparisons are like this (we use the inverse of the carry flag in stead of the sign flag):

a==b subf r0, r1; jpz ...

a!=b subf r0, r1; jpnz ...

a<b subf r0, r1; jpnc ...

a>b subf r1, r0; jpnc ...

a<=b subf r1, r0; jpc ...

a>=b subf r0, r1; jpc ...

In the C programming language the result of a comparison can be stored in a Boolean variable. This is an integer that can be zero (false) or nonzero (true). The Moncky-3 processor has instructions to save the flags in a register. The value 'true' will be stored as 0xFFFF. In the following example the result of the comparison is put in register r2.

For signed integers the following code can be used:

a==b subf r0, r1; sz r2

a!=b subf r0, r1; snz r2

a<b subf r0, r1; ss r2

a>b subf r1, r0; ss r2

a<=b subf r1, r0; sns r2

a>=b subf r0, r1; sns r2

For unsigned numbers the following code can be used:

a==b subf r0, r1; sz r2

a!=b subf r0, r1; snz r2

a<b subf r0, r1; snc r2

a>b subf r1, r0; snc r2

a<=b subf r1, r0; sc r2

a>=b subf r0, r1; sc r2

7.7 Avoiding conditional jumps

Saving the status of a flag in a register has another advantage. It allows to avoid conditional jumps. This can be used for the following kind of expression:

```
c = (a==b)?d:e
```

The idea is to set c to the value of d if the condition is met. Otherwise c should be set to e. Let's assume the values of a, b, c, d, and e are kept in registers r0, r1, r2, r3, and r4 respectively. Then the following code will execute the previous statement:

```
subf r0, r1 ; compare a and b
li r5, :else
lih r5, ::else
jpnz [r5]
set r2, r3
li r5, :endif
lih r5, ::endif
jp [r5]
:else
set r2, r4
:endif
```

Now look at the following code. It will have the same result, but is does not contain any conditional jumps:

```
subf r0, r1 ; compare a and b
sz r5       ; save the zero flag (r5 is now 0 or 0xFFFF)
snz r6      ; save the inverse zero flag
and r5, r3  ; r5 is now 0 or d
and r6, r4  ; r6 is now 0 or e
or r5, r6   ; r5 now contains the right value
set r2, r5  ; save the result in the right register
```

Remark that the saved flag is used as a mask. When 'and'-ing this mask to a value, the result is 0 or the value. This code doesn't need any labels or conditional jumps and it will always execute in the same amount of clock cycles (which is less than the worst case of the previous code).

The latter code is also very interesting when the processor would have been pipelined since no branch prediction needs to be done.

7.8 Absolute value

Finding the absolute value of a 2-complement number can also be done without any conditional jumps. For example, the following code calculates the absolute value of r0:

```
set r1, r0   ; save r0 to r1
ashri r1, 15 ; extend the sign bit (r1 is now 0 or 0xFFFF)
xor r0, r1   ; invert all bits in r0 if r1==0xFFFF
sub r0, r1   ; add one if r1==0xFFFF
```

Chapter 8

Firmware

The firmware consists of 2 parts: the code in the ROM and the code that is read from the SD card. When the computer boots, the code in ROM is executed and replaced by the contents of the file 'MONCKYOS' on the first partition of the SD card.

8.1 ROM boot sequence

When the computer starts, the ROM shadowing is activated. The processor will execute the following steps:

- show a splash screen
- find and initialise an SD card
- find and initialise the first partition (<32MB) on the disk
- find and read the file 'MONCKYOS' to memory at location 0x0000
- jump to a specific location (0x0008) and disable ROM shadowing

Errors might occur during this process. In order to keep the ROM footprint small only error codes are shown. The following codes can be generated:

0001 No SD card was found (the card does not respond to the reset (0) command)

0002 SD card error (non compatible voltage range after command 8)

0003 SD read error (the card does not respond to the read block (17) command)

0004 SD2 card error (the card did not respond to command 58)

0005 SD card error (the card does not go to ready state after command 41)

0006 SD read timeout (the card does not start sending data after requesting a block)

0007 SD read error (the card does not start the data with the right byte 0xFE)

0008 FAT not formatted (the MBR does not contain the right signature bytes 0x55 and 0xAA at the end)

0009 FAT first logical partition too big (at the moment partitions must be strictly smaller than 32 MiB)

000A FAT invalid sector size (only sectors of 512 bytes are supported)

000B FAT invalid number of FATs (there must be exactly 2 FAT tables on disk)

000C FAT OS not found (the file 'MONCKYOS' could not be found in the root folder of the disk)

8.2 Firmware

After the boot sequence is done, control is handed over to the code in 'MONCKYOS'. This can be whatever the programmer wants it to be. However, some functionality has already been implemented and is available in the repository on Gitlab in the 'firmware' folder. These files can be linked to your own code by the assembler.

8.2.1 System setup

The system starts at address 0x0000. The following commands are set here ('system.asm'):

```
.def :startOfStack 0xFFFE
.def :system_screenConfig 0x7D00
.org 0x0000
        di
        li sp, :startOfStack
        lih sp, ::startOfStack
        li r0, :start_system
        lih r0, ::start_system
        jp [r0]
.org 0x0008
        reset
        reset
        reset
        reset
        reset
        reset
        reset
        reset
.org 0x0010
        push r0 ; save all registers and flags
        push r1
        push r2
        sflags r2
        li r0, :system_screenConfig
        lih r0, ::system_screenConfig
        in r1, (r0)
        xori r1, 2 ; toggle bit
        out r1, (r0)
        rflags r2 ; restore flags and registers
        pop r2
        pop r1
        pop r0
        reti
```

Basically the processor will set the top of stack and jump to the start of the system, which is located further away (at least after the interrupt handler).

When the ROM disables the ROM shadowing, it is executing the following instructions:

```
.org 0x0008
li r0, 0xFF
lih r0, 0xFF
li r1, 0xAA
lih r1, 0xAA
st r1, (r0)
reset
```

This means that the processor will execute the 'reset' instruction at address 0x000D. In order for this to work, there should also be a reset instruction in this location in the operating system. This is why there are reset instructions from address 0x0008 till 0x000F.

At address 0x0010 the hardware interrupt is located. In the default code, the interrupt handler will toggle the second bit in the screen configuration. This bit is not used by the hardware. It can be used to detect the vertical retrace in software.

Important: the interrupt handler must always save all registers that are used and also the flags! Otherwise the code will not run properly when the processor returns to the point the interrupt occurred!

8.2.2 Graphics

The graphics library ('graphics.asm') contains functionality to control the pixels on the screen.

The following subroutines are available:

- void graphics_init(): initialises the screen to default values (does not clear the screen)
- void graphics_setBgCol(uint8 col): sets the background colour to the specified value
- void graphics_clrscr(): clears the screen (with the background colour)
- void graphics_setCol(uint8 col): sets the current colour (used by plot and line)
- void graphics_setScreenBuffer(uint1 buffer): sets the current buffer to be displayed on the screen (happens after a vertical retrace)
- void graphics_waitVertRetrace(): waits until the next vertical retrace happened
- void graphics_setDrawBuffer(uint1 buffer): sets the current buffer to draw on (used by plot and line)
- uint1 graphics_getDrawBuffer(): gets the current buffer to draw on
- void graphics_plot(uint16 x, uint16 y): plots a point on the given coordinates with the current colour on the current buffer
- void graphics_line(uint16 x0, uint16 y0, uint16 x1, uint16 y1): draws a line between 2 given points with the current colour on the current buffer
- void graphics_scrollUp(uint16 amount): scrolls the screen up by the given amount of lines. The bottom lines are filled with the background colour

8.2.3 Text

The text library ('text.asm') contains functionality to draw text on the screen. It uses the graphics library to draw pixels. Characters are drawn in a 8x8 pixel grid. The library currently only supports ASCII characters. The first 32 characters are special:

- 0x01: a heart character (to show the authors love to his partner :-))
- 0x02: an 8x8 square that can be used to display a cursor
- 0x03, 0x04, 0x05, 0x06, 0x07, 0x0B, 0x0C, 0x0E, 0x0F: characters to draw the BigBat logo
- 0x08: back space
- 0x09: tab
- 0x0A: new line
- 0x0D: carriage return
- 0x11: arrow left
- 0x12: arrow right
- 0x13: arrow up
- 0x14: arrow down

The following subroutines are available:

- void text_init(): sets cursor at (0,0)
- void text_setCursor(uint8 x, uint8 y): sets the cursor at the given coordinates (x<80 and y<25)
- void text_drawCursor(): draws the cursor at the current cursor position while saving the overwritten part in memory
- void text_eraseCursor(): restores the part of the screen that was overwritten by the cursor
- void text_drawChar(uint16 x, uint16 y, uint8 char): draws a character (0-127) at the given coordinates (x<320 and y<200)
- void text_printChar(uint8 char): draws a character at the actual cursor position. This function will also move the cursor and scroll the screen up when necessary. It recognises the arrow left, right, up, and down commands as well as the backspace and newline characters.
- void text_printString16(uint16* string): prints a null-terminated string on the screen. There is one 16-bit word per character.
- void text_printString(uint8* string): prints a null-terminated string on the screen. There are two characters per 16-bit word: the LSB is the first character and the MSB is the second.
- void text_printHex(uint16 value): prints the given 16-bit value on the screen in 4 hexadecimal digits (with leading zeroes)

8.2.4 Keyboard

The keyboard library ('keyboard.asm') contains functionality to read the next key from the keyboard. It interacts with the hardware keyboard buffer. The keyboard buffer will emit 'key codes' which are 10 bit values. The 9 least significant bits are the value of the key and the 10th bit indicates if the key was released. The 8th bit indicates an 'extended key code'.

When a key code is translated to ASCII the following special keys are defined:

- 0x01: F1
- 0x02: F2
- 0x03: F3
- 0x04: F4
- 0x05: F5
- 0x06: F6
- 0x07: F7
- 0x08: backspace
- 0x09: tab
- 0x0A: enter
- 0x0B: F8
- 0x0C: F9
- 0x0D: carriage return
- 0x0E: F10
- 0x0F: F11
- 0x10: F12
- 0x11: arrow left
- 0x12: arrow right
- 0x13: arrow up
- 0x14: arrow down
- 0x15: delete
- 0x16: insert
- 0x17: page up
- 0x18: page down
- 0x19: home
- 0x1A: end

- 0x1B: esc
- 0x1C: menu

The following subroutines are available:

- void keyboard_init(): will erase the keyboard buffer in the keyboard controller
- uint16 keyboard_readKeyCode(): will return the key code of the next key in the buffer. If no key was pressed, zero is returned.
- uint16 keyboard_waitKey(uint16 timeout): will wait until a key is pressed. It will return 0 if a timeout occurs (time is in milliseconds). If the timeout is 0 the computer will wait indefinitely.
- uint16 keyboard_toChar(uint16 keycode): will return the ASCII code that corresponds with the given key code. It translates key codes to ASCII. If the shift key was pressed the corresponding value will be altered accordingly.
- uint16 keyboard_readChar(): will return the ASCII code that corresponds with the key pressed on the keyboard (this is readKeyCode followed by toChar)
- uint16 keyboard_waitChar(uint16 timeout): wait until a key is pressed and return the corresponding ASCII code (this is waitKey followed by toChar)

8.2.5 Utilities

The utilities library ('utilities.asm') contains some extra utilities that can be handy.

The following subroutines are available:

- void util_log(message, numberOfParameters): this is a special function. It does not follow the normal calling convention. The parameters are passed as r1 and r2 respectively. The function is called right after the 'push bp, set bp, sp' sequence. It will display the current time (milliseconds), a message and all parameters on screen. A programmer can use this to debug software
- void util_sleepMillis(uint16 time): this function will wait for the specified amount of milliseconds. This is a blocking wait.
- void util_error(uint16 code): this function will display an error on the screen, with the given code in hexadecimal

8.2.6 SPI

The SPI library ('spi.asm') contains functionality to initialise and use the SPI interface.

The following subroutines are available:

- void spi_init(): this will set the slave select pins to high, SPI mode to 0, and speed to maximum.
- void spi_enable(uint8 slave): this will put the selected slave select pin to 0 and the others to 1

- void spi_disable(): this will put all slave select pins to 1
- void spi_setSpeed(uint8 speed): this will put the speed of the SPI interface to the given value (the value is calculated as indicated in section 6.8).
- uint8 spi_transfer(uint8 byte): this will transfer a byte to the SPI slave. The return value is the received value.

8.2.7 SD card

The SD card library ('sd.asm') contains functionality to initialise and read blocks from an SD card. Writing is currently not implemented.

The SD card is supposed to be connected to the SPI interface. A nice introduction to SD cards can be found at: http://www.rjhcoding.com/avrc-sd-interface-1.php.

The code is derived from the following C-code:

```
#define SS 7
#define MISO 10
#define MOSI 8
#define SCK 9
#define SD_SUCCESS  0
#define SD_ERROR_CANNOT_GO_TO_IDLE_STATE 1
#define SD_ERROR_CANNOT_DETECT_TYPE 2
#define SD_ERROR_TIMEOUT 3
#define SD_ERROR_CANNOT_GET_OCR 4
#define SD_ERROR_CANNOT_SET_BLOCK_SIZE 5
#define SD_ERROR_CANNOT_READ 6
uint8 sd_error;
uint8 sd_type;
uint8 sd_result[5];
void sd_disable() { spi_transfer(0xFF); spi_disable(); spi_transfer(0xFF); }
void sd_enable() { spi_transfer(0xFF); spi_enable(0); spi_transfer(0xFF); }
void sd_command(uint8 cmd, uint32 arg, uint8 crc) {
    spi_transfer(cmd|0x40);
    spi_transfer(arg >> 24);
    spi_transfer((arg >> 16) & 0xFF);
    spi_transfer((arg >> 8) & 0xFF);
    spi_transfer(arg & 0xFF);
    spi_transfer(crc|0x01);
}
void sd_powerUpSeq() {
    sd_disable();
    util_sleepMillis(1);
    spi_transfer(0xFF);
    spi_transfer(0xFF);
    spi_transfer(0xFF);
    spi_transfer(0xFF);
    spi_transfer(0xFF);
    spi_transfer(0xFF);
    spi_transfer(0xFF);
    spi_transfer(0xFF);
```

```
    spi_transfer(0xFF);
    spi_transfer(0xFF);
}
uint8 sd_readRes1() {
    uint8 i = 0, res1;
    while((res1 = spi_transfer(0xFF)) == 0xFF) {
        i++;
        if (i > 8) break;
    }
    return res1;
}
uint8 sd_readUntil(uint8 expected, uint16 timeout) {
  uint8 res1;
  uint16 t1 = util_msec_LSB;
  while((res1 = spi_transfer(0xFF)) != expected) {
    if ((util_msec_LSB-t1)>timeout) return 0xFF;
  }
  return res1;
}
uint8 sd_goIdleState() {
    sd_enable();
    sd_command(0, 0, 0x94);
    uint8 res1 = sd_readRes1();
    sd_disable();
    return res1;
}
void sd_readRes3_7(uint8 *res) {
    res[0] = sd_readRes1();
    if (res[0] > 1) return;
    res[1] = spi_transfer(0xFF);
    res[2] = spi_transfer(0xFF);
    res[3] = spi_transfer(0xFF);
    res[4] = spi_transfer(0xFF);
}
void sd_sendIfCond(uint8 *res) {
    sd_enable();
    sd_command(8, 0x0000001AA, 0x86);
    sd_readRes3_7(res);
    sd_disable();
}
void sd_readOCR(uint8 *res) {
    sd_enable();
    sd_command(58, 0, 0);
    sd_readRes3_7(res);
    sd_disable();
}
uint8 sd_sendApp() {
    sd_enable();
    sd_command(55, 0, 0);
    uint8 res1 = sd_readRes1();
    sd_disable();
```

```
    return res1;
}
uint8 sd_sendOpCond() {
    sd_enable();
    sd_command(41, (sd_type==2?0x40000000:0), 0);
    uint8_t res1 = sd_readRes1();
    sd_disable();
    return res1;
}
uint8 sd_sendBlockSize() {
  sd_enable();
  sd_command(16, 512, 0);
  uint8_t res1 = sd_readRes1();
  sd_disable();
  return res1;
}
uint8 sd_init() {
    sd_type = 0;
    uint8 cmdAttempts = 0;
    sd_powerUpSeq();
    while((sd_result[0] = sd_goIdleState()) != 0x01) {
        cmdAttempts++;
        if (cmdAttempts > 10) {
          sd_error = sd_result[0];
          return SD_ERROR_CANNOT_GO_TO_IDLE_STATE;
        }
    }
    sd_sendIfCond(sd_result);
    sd_type = (result[0] & 4 == 0x04)?1:2;
    if (sd_type == 2 && sd_result[4] != 0xAA) {
        sd_error = sd_result[0];
        return SD_ERROR_CANNOT_DETECT_TYPE;
    }     cmdAttempts = 0;
    do {
        if (cmdAttempts > 100) return SD_ERROR_TIMEOUT;
        sd_result[0] = sd_sendApp();
        if(sd_result[0] < 2) {
            sd_result[0] = sd_sendOpCond();
        }
        util_sleepMillis(10);
        cmdAttempts++;
    } while(sd_result[0] != 0);
    sd_readOCR(sd_result);
    if (!(sd_result[1] & 0x80)) return SD_ERROR_CANNOT_GET_OCR;
    sd_type = (sd_result[1] & 0xC0 == 0xC0)?3:sd_type;
    uint8 res = sd_sendBlockSize();
    if (res != 0) return SD_ERROR_CANNOT_SET_BLOCK_SIZE; else return SD_SUCCESS;
}
uint8 sd_read(uint32 blockNumber, uint16 *destination) {
  sd_enable();
  if (sd_type != 3) blockNumber <<= 9;
```

```
  sd_command(17, blockNumber, 0);
  uint8 res1 = sd_readUntil(0xFE, 100);
  if (res1 != 0xFE) return SD_ERROR_CANNOT_READ;
  for(uint16 i = 0; i < 256; i++) {
    *destination = spi_transfer(0xFF) | (spi_transfer(0xFF) << 8);
    destination++;
  }
  // skip CRC
  spi_transfer(0xFF);
  spi_transfer(0xFF);
  sd_disable();
  return SD_SUCCESS;
}
```

The following subroutines are available (only the publicly availabe are shown here):

- uint8 sd_init(): this will detect and initialise the card. If anything goes wrong, an error code is returned. The error codes differ from the ones generated by the ROM and can be seen in the C-file above. This routine should be called after an SD card is inserted.
- uint8 sd_read(uint32 blockNumber, uint16* destination, bool io): will read the specified block (512 bytes) into memory to the specified location. If the Boolean 'io' is true, the data is written to i/o memory in stead of normal RAM. The destination should point to an array containing 256 16-bit words. The words are read LSB first. If an error occurs the code is returned. On success the return value is equal to 0.

8.2.8 FAT

The FAT library ('fat.asm') contains functionality to interact with a FAT16 file system, located on the first partition of an SD card. This partition must be strictly smaller than 32MB.

The code is derived from the following C-code:

```
#define FAT_OK                 0
#define FAT_ERROR_NOT_FORMATTED       1
#define FAT_ERROR_PARTITION_TOO_BIG      2
#define FAT_ERROR_INVALID_SECTOR_SIZE    3
#define FAT_ERROR_INVALID_NUMBER_OF_FATS 4
#define FAT_ERROR_FILE_NOT_FOUND         5
#define FAT_END_OF_FILE                  6
uint32 fat_startPartition;
uint32 fat_lengthPartition;
uint16 fat_fatPosition;
uint32 fat_rootFolderPosition;
uint16 fat_numberOfSectorsForRootFolder;
uint16 fat_fileStart;
uint32 fat_fileLength;
uint16 fat_currentClusterNumber;
uint8 fat_sectorsPerCluster;
uint8 fat_currentSectorNumber;
uint16 fat_sectorsPerFat;
uint16 fat_sector[256];
```

```
uint8 fat_findAndCheckPartition() {
    fat_startPartition = 0;
    fat_lengthPartition = 0;
    sd_read(0, fat_sector); // read MBR
    uint16 signature = fat_sector[255];
    if (signature != 0xAA55) return FAT_ERROR_NOT_FORMATTED;
    fat_startPartition = fat_sector[0xE3] + fat_sector[0xE4] << 16;
    fat_lengthPartition = fat_sector[0xE5] + fat_sector[0xE6] << 16;
    if (fat_lengthPartition >= 65536) return FAT_ERROR_PARTITION_TOO_BIG;
    return FAT_OK;
}
uint8 fat_findAndCheckFAT() {
    fat_fatPosition = 0;
    fat_rootFolderPosition = 0;
    fat_numberOfSectorsForRootFolder = 0;
    sd_read(fat_startPartition, fat_sector); // VBR
    uint16 bytesPerSector = (fat_sector[5] >> 8) | (fat_sector[6] & 0xFF) << 8;
    if (bytesPerSector != 512) return FAT_ERROR_INVALID_SECTOR_SIZE;
    uint8 numberOfFATs = fat_sector[8] & 0xFF;
    if (numberOfFATs != 2) return FAT_ERROR_INVALID_NUMBER_OF_FATS;
    fat_sectorsPerFat = fat_sector[11];
    fat_fatPosition = fat_sector[7];
    fat_rootFolderPosition = fat_startPartition + fat_fatPosition + 2 * fat_sectorsPerFa
    uint16 maxNumberOfRootDirEntries = (fat_sector[8] >> 8) | (fat_sector[9] & 0xFF) <<
    fat_numberOfSectorsForRootFolder = maxNumberOfRootDirEntries / 16;
    fat_sectorsPerCluster = fat_sector[6] >> 8;
    return FAT_OK;
}
uint8 fat_openFile(char filename[11]) {
    fat_fileStart = 0;
    fat_fileLength = 0;
    fat_currentSectorNumber = 0;
    fat_currentClusterNumber = 0xFFFF; // end of file
    uint16 i = 0;
    while (i < fat_numberOfSectorsForRootFolder) {
        sd_read(rootFolderPosition + i, fat_sector); // part of root folder
        uint8 entry = 0;
        while (entry < 16) {
            uint16 position = entry * 16;
            char* name = fat_sector + position;
            if (strcmp(name, filename, 11)==0) {
                fat_currentClusterNumber = fat_fileStart = fat_sector[position + 13];
                fat_fileLength = fat_sector[position + 14] + fat_sector[position + 15] <
                return FAT_OK;
            }
            entry++;
        }
        i++;
    }
    return FAT_ERROR_FILE_NOT_FOUND;
}
```

```
uint8 fat_readNextFileSector(uint16 buffer[256]) {
    if (fat_currentClusterNumber >= 0xFFF8) return FAT_END_OF_FILE;
    sd_read(fat_rootFolderPosition + fat_numberOfSectorsForRootFolder
            + (fat_currentClusterNumber-2)*fat_sectorsPerCluster
            + fat_currentSectorNumber, buffer);
            // weird thing: FAT cluster numbers start at 2...
    fat_currentSectorNumber++;
    if (fat_currentSectorNumber >= fat_sectorsPerCluster) {
        fat_currentSectorNumber = 0;
        uint16 fatSectorNumber = fat_currentClusterNumber >> 8;
        uint16 fatOffset = fat_currentClusterNumber & 0xFF;
        sd_read(fat_startPartition + fat_fatPosition + fat_fatSectorNumber, fat_sector);
        fat_currentClusterNumber = fat_sector[fatOffset];
    }
    return FAT_OK;
}
```

The following (public) subroutines are available:

- uint8 init(): will find and check the first partition and will localise the FAT16 file system. It returns FAT_OK if everything is ok, otherwise an error code is returned (see code above)
- uint8 fat_openFile(char filename[11]): will try to find the file with the given filename (11 characters are expected). Each two characters are expected to be in a 16-bit word in memory (LSB first). If less than 11 characters are present, spaces should be added so the length is always 11. The string must not be null-terminated. If the file was not found, an error code is returned. On a FAT-16 filesystem all filenames are in capitals.
- uint8 fat_readNextFileSector(uint16 buffer[256], boolean io): will read the next file sector (512 bytes, 256 words) into the given location (the boolean indicates if the buffer is located in i/o space). If there are more sectors available, FAT_OK is returned. A return value FAT_END_OF_FILE means the end of the file has been reached (the last sector has just been read).

8.2.9 Integer math

to be implemented

8.2.10 Floating point math

to be implemented

8.2.11 String functions

to be implemented

8.2.12 Interpreter

to be implemented