Practicum Manual

Handbook for the RUN2223 CPU

1 Introduction

This manual describes the RUN2223 CPU, a 32-bit CPU designed specifically for the course Processors. In the practical assignment for this course, your task will be to create an implementation of the RUN2223 CPU in Digital, according to the specifications in this manual.

In this manual you will find:

- a specification of the input and output interfaces of the RUN2223 CPU, which enable it to communicate with the rest of the computer
- a description of the instruction cycle of the CPU
- a general overview of the architecture of the CPU, and how it is divided into subcomponents
- a specification of the assembly language instructions, and the corresponding machine codes, of the RUN2223 CPU

Occasionally, the descriptions in this manual will deliberately be incomplete. In these cases it is up to you to make sensible design choices.

How to use this manual

This manual contains a lot of information, and we do not expect you to understand everything in one go. Nevertheless, we recommend that you read through the complete manual at least once, so that you know where to find things.

For the data flow diagrams homework exercises, you will need information from sections 2 and 3.

The practical assignment itself will be published as a (short) separate document which should be read alongside this manual. We will also provide you with several Digital template files on which to base your solution.

2 Interfaces of the CPU

The RUN2223 CPU communicates with the other parts of the computer (e.g. Main Memory, I/O Devices) through a *bus*. The bus consists of a set of wires that all devices are connected to, together with a protocol for how communication takes place. In the RUN2223 CPU the bus is further subdivided into a data bus, an address bus, and control signals that indicate when the CPU wants to read or write data. These are specified in detail below.

The RUN2223 CPU has the following inputs:

Databus$_{\texttt{in}}$	A 32-bit databus for data input, coming from the RAM or other devices such as the keyboard.
Clock	A 1-bit wire that is driven by the central clock of the computer. If the CPU wants to write output (to RAM or other devices) then this happens on the next rising edge of the clock. If the CPU wants to read data (from RAM or other devices) then it assumes that the data is available at the next rising edge of the clock.

The RUN2223 CPU has the following outputs:

Addressbus	A 32-bit addressbus that provides the address that the RUN2223 CPU wants to read from or write to. The computer itself will inspect the address to determine which device (e.g. RAM or Terminal) should handle the read/write request. This is implemented outside of the CPU, so the RUN2223 CPU itself treats each address equally.
Databus$_{\texttt{out}}$	A 32-bit databus with data that is to be written.
WE	Write enable. A 1-bit wire that is 1, if and only if, the data provided in **Databus**$_{\texttt{out}}$ has to be written to an external device, such as RAM.
RE	Read enable. A 1-bit wire that is 1, if and only if, the RUN2223 CPU wants to read data from an external device, such as RAM.
Halted	A 1-bit wire that is 1, if and only if, the RUN2223 CPU has halted.

3 Global Overview

The RUN2223 CPU has 16 registers, each 32 bits wide, called $R_0, R_1, \ldots R_{15}$. The first register R_0 always contains the value 0. Writing to this register thus has no effect. Register R_{13} can be used as a frame pointer (FP). Register R_{14} is the stack pointer (SP). Register R_{15} is the program counter (PC).

3.1 Instruction Cycle

The instruction cycle of the RUN2223 CPU consists of four phases:

Fetch	the CPU takes the address stored in the PC, puts it on the addressbus, and signals a memory read. This causes a machine code to be put on the databus, which is then stored in the instruction register. At the end of the phase, the program counter is incremented by 4.
Test	the CPU tests whether the preconditions for the current instruction have been met. If yes, then it will start the **D+Execute** phase. If not, then it will skip that phase and start with the next **Fetch** phase.
D+Execute	the CPU decodes the current instruction and executes it. At the end of this phase, depending on the instruction, data is stored in a register or written to an external device.
Pause	the CPU is idle, waiting for the clock to go back to 0, at which point the CPU will be in the **Fetch** phase again.

Example. Let's consider a possible assembly instruction

```
ADDf.NZ RA, RB, RD
```

The instruction contains the opcode for addition, plus the addresses of the source registers A and B and the destination register D. It also contains a condition that provides preconditions: if these are not met, the instruction will not be executed. In this example, the precondition NZ means that the Zero flag must be false. During the **Fetch** phase, the instruction is read. In the **Test** phase, the condition is evaluated. If the test succeeds, then the **D+Execute** phase will start. After execution, the following holds:

$$R_D := R_A + R_B$$

Also, the flags will have been set.

3.2 Architecture of the CPU

The CPU is built up of several subcomponents, each of which has a specific role in the instruction cycle. For two of these components, the *Timer* and the *Register bank*, we give a complete specification of the input and output interfaces below. In addition, we will provide you with an implementation of these components in Digital, which you must include directly in your own design.

For the remaining four components, we provide complete interfaces, except for the `InstructionDecoder`. The data flow diagrams homework exercises will help you determine which inputs and outputs need to be added to this component (if any), and how all the components should be connected to implement the CPU.

Timer	The timer is used to guide the RUN2223 CPU through the execution phases. It has outputs **`Execute`** and **`Fetch`** that are true if and only if the processor is in that phase. Additionally, it controls when the rest of the CPU is **`Halted`**.
Register bank	Contains the 16 registers. Input $\texttt{Data}_{\texttt{in}}$ contains the 32-bit value that is to be stored in the destination register. The 4-bit inputs $\texttt{Reg}_{\texttt{A}}$, $\texttt{Reg}_{\texttt{B}}$, and $\texttt{Reg}_{\texttt{dest}}$ contain the addresses of the source registers A and B and the destination register. Finally, the register bank uses the **`Clock`**. As output, the register bank provides $\texttt{Data}_{\texttt{A}}$ and $\texttt{Data}_{\texttt{B}}$ which hold the contents of the source registers A and B.
Instruction decoder	The instruction decoder contains an instruction register. The instruction in that register is decoded into various parts, such as the source and destination registers, its condition and constants. During the **Fetch** phase, an instruction is loaded into this register. During the **Execute** phase, control signals are provided for other components. It is up to you to determine which.
Flag reg bank	This register bank stores the four flags using registers. It has an input per flag, a set enabled input, and an output per flag.
Tester	The tester decides whether the precondition of the current instruction holds. It provides as output a 1-bit wire **`TestSucceeds`**. This result is sent to the timer.
ALU	The ALU takes as input two 32-bit inputs **`A`** and **`B`** and a 3-bit opcode. It provides a 32-bit output called **`Result`**. The ALU provides four flags: Negative, Overflow, Zero, Carry.

4 Assembly and machine codes

4.1 Assembly Instructions

Appendix A provides the assembly instruction set. The first column shows the instruction name. Each assembly instruction has a condition. If no explicit condition is specified in the assembly instruction, the default condition of `.T` is used. Some instructions have an f-variant, for example, the `ADD` has a variant called `ADDf`. The f-variant performs the calculation, but also sets the flags. The regular variant does not set flags.

Furthermore, each assembly instruction has 0 or more arguments. Column 3 shows these arguments, which may include:

- the 4-bit addresses of source registers A, B, and the destination register D
- a 10-bit signed constant in two's-complement, denoted *c10*
- a 22-bit signed constant in two's-complement, denoted *c22*
- either the address of a source register A, *or* a 10-bit signed constant, denoted $A/c10$

Column 4 shows the opcodes for the arithmetic operations computed by the ALU. For each arithmetic operation, the first argument is either the address of a register, or a 10-bit signed constant. For example, the assembly instruction

```
ADDf.T R15, R15, R2
```

will double the value stored in R_{15} and store the result in R_2. In contrast, assembly instruction

```
ADDf.T 15, R15, R2
```

will add 15 to the value stored in R_{15} and store the result in R_2.

Column 5 provides the specification of what the computation should do. Not all assembly instructions have machine code. Some assembly is translated to other assembly. For example:

`NOT.cond A, D` $\equiv$ `XOR.cond -1, A, D`

Thus, by implementing `XOR`, the `NOT` instruction is implemented as well. Column 6 shows for which instructions there exists a translation.

4.2 Machine Codes

Appendix B shows the machine code corresponding to the assembly instructions. All machine codes have the condition in the four least significant bits. Registers are addressed using 4 bits in regular binary format. Constants are signed and in two's-complement. Other machine codes than these are prohibited: for these codes the behavior is unspecified.

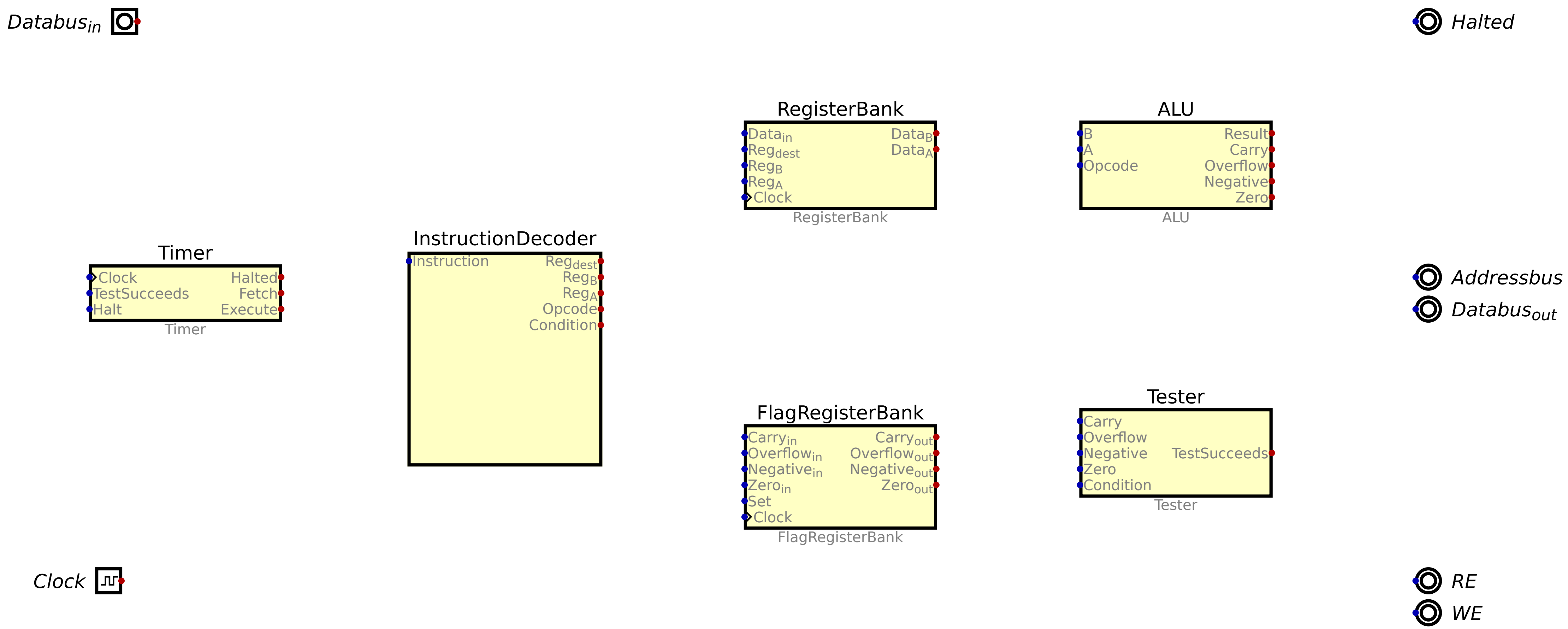

Figure 1: Components of the RUN2223 CPU.

Appendix A: assembly code

Instruction	f-variant?	arguments	opcode	Specification	Translated?
`ADD.cond`	yes	$A/c10$, B, D	110	Addition: $D := A + B$	no
`AND.cond`	yes	$A/c10$, B, D	010	Bitwise and: $D := A \,\&\, B$	no
`SHL.cond`	yes	$A/c10$, B, D	011	Logical shift left: $D := B \ll A$	no
`SHR.cond`	yes	$A/c10$, B, D	100	Logical shift right: $D := B \gg A$	no
`CMPf.cond`	only	$A/c10$, $B/c10$		Compare: compute $A - B$ and set flags, but ignore result. Two constants is not allowed.	yes
`HALT.cond`	no			Halt: stop the CPU	no
`JUMP.cond`	no	$c10$		Jump to the given address	yes
`LOADHI.cond`	no	$c22$, D		Load $c22$ into highest bits and put result in D	no
`MOVE.cond`	yes	$A/c10$, D		Move: copy value in A to D	yes
`NEG.cond`	yes	A, D		Negate: $D := -A$	yes
`NOP`	no			No operation	yes
`NOT.cond`	yes	A, D		Bitwise not: $D := \sim A$	yes
`OR.cond`	yes	$A/c10$, B, D	000	Bitwise or: $D := A \mid B$	no
`READ.cond`	no	$[B + c10]$, D		Read from external device at address $c10 + B$, and store the result in D	no
`ROL.cond`	yes	$A/c10$, B, D	101	Rotate left: $D := B$, rotated A bits to the left	no
`ROR.cond`	yes	$c10$, B, D		Rotate right: $D := B$, rotated $c10$ bits to the right	yes
`SUB.cond`	yes	$A/c10$, B, D	111	Subtract: $D := A - B$	no
`WRITE.cond`	no	A, $[B + c10]$		Write to external device at address $c10 + B$, writing the value stored in A	no
`XOR.cond`	yes	$A/c10$, B, D	001	Bitwise exclusive-or: $D := A \oplus B$	no

Table 1: Assembly instruction set

bitcode	**suffix**	**meaning**	**precondition**
0000	.Z or .E	zero, equal	Z
0001	.NZ or .NE	not zero, not equal	!Z
0010	.C or .GEU	carry, greater or equal (unsigned)	C
0011	.NC or .LU	not carry, less (unsigned)	!C
0100	.N	negative	N
0101	.NN	not negative	!N
0110	.O	overflow	O
0111	.NO	not overflow	!O
1000	.GU	greater (unsigned)	C & !Z
1001	.LEU	less or equal (unsigned)	Z | !C
1010	.GE	greater or equal (signed)	N == O
1011	.L	less (signed)	N != O
1100	.G	greater (signed)	(N == O) & !Z
1101	.LE	less or equal (signed)	(N != O) | Z
1110	.F	false	0
1111	.T (default)	true	1

Table 2: Conditions

Be aware that many of the conditions (such as .GU) assume the flags were set using a `CMPf` instruction. The instruction `XORf R1, R2, R3` for example will always set the carry flag to false, regardless of the values of R_1 or R_2. As the carry flag is false, the .LU condition will be true, but that does not necessarily mean that R_1 is less than R_2. In this example .NC would still have the correct *semantic* (namely only executing if the carry flag is not set), whereas .LU does *not*, even though their preconditions are identical. Make sure you use conditions in an appropriate context during your assembly programming, so that they have a correct semantic meaning.

Appendix B: machine code

1	`HALT`	When the CPU has executed a `HALT` , it will do nothing anymore. If input **`Halt`** of the timer is set to 1 during the **D+Execute** phase, it assumes that a `HALT` instruction is being executed. From that moment on, the timer will set outputs **`Fetch`** and **`Execute`** to 0 and **`Halted`** to 1.
2	`WRITE`	A write computes the external address to write to by adding the given offset to the address stored in the given register. Use the ALU for this computation; the result is to be sent to the **`Addressbus`**, instead of a register. The value to write is sent to $\texttt{\textbf{Databus}}_{\texttt{\textbf{out}}}$.
3	`READ`	A read computes the external address to read from by adding the given offset to the address stored in the given register. Use the ALU for this computation; the result is to be sent to the **`Addressbus`**, instead of a register. The input on the $\texttt{\textbf{Databus}}_{\texttt{\textbf{in}}}$ is sent to the registerbank.
4	`LOADHI`	The `LOADHI` serves to load any constant – up to 32-bit – within two instructions. First, use `LOADHI` to set the 22 most significant bits, subsequently use `ADD` to set the 10 least significant bits. Note that this requires that after the first step, the 10 least significant bits are set to zero.
5	`ARITH3`	Any arithmetic operation. The field `flg` is true, if and only if, the operation sets the flags. The operation takes as input the value stored in register A. The opcodes and flags are given in Table 3.
6	`ARITH2`	Any arithmetic operation. The field `flg` is true, if and only if, the operation sets the flags. The operation takes as input the 10-bit constant value `c10`, sign extended to 32 bits. The opcodes and flags are given in Table 3.

`OR`	000	$C := 0$ $O := 0$	$N :=$ bit 31 of result $Z := 1$ iff result $= 0$
`XOR`	001		
`AND`	010		
`SHL`	011	$C :=$ see Appendix C $O :=$ see Appendix C	
`SHR`	100		
`ROL`	101		
`ADD`	110		
`SUB`	111		

Table 3: ALU opcodes and flags

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	cond			
2	1	1	1	0	0	0	c10										A				B				0	0	0	0	cond			
3	1	1	0	0	0	0	c10										0	0	0	0	B				D				cond			
4	1	0	c22																						D				cond			
5	0	opcode			flg	1	0	0	0	0	0	0	0	0	0	0	A				B				D				cond			
6	0	opcode			flg	0	c10										0	0	0	0	B				D				cond			

Table 4: Machine code

Appendix C: carry and overflow flags

Addition

While performing a 32-bit addition, the carry flag is used in the context of unsigned operands. It is set if, and only if, the result is too large to fit into 32-bits, and an additional 33rd bit is required. The `Adder` component in Digital has a carry output with this exact semantic, and is recommended for implementing the ADD instruction.

While performing a 32-bit addition, the overflow flag is used in the context of signed operands. It is set if, and only if, the result does not fit into 32-bits. There are exactly two cases where this can occur, determined by looking at the sign of the operands and the result:

$$\begin{aligned}\text{Positive} &+ \text{Positive} = \text{Negative}\\ \text{Negative} &+ \text{Negative} = \text{Positive}\end{aligned}$$

The `Adder` component does not have an overflow output, meaning you will have to implement this logic yourselves.

Subtraction

While performing a 32-bit subtraction, the carry flag is used in the context of unsigned operands. Unfortunately, unlike for addition, the carry flag is not well defined. There are two common conventions:

1. Subtract with borrow. In this convention, the carry (borrow) flag is set when computing $a - b$ if, and only if, $a < b$. This convention is used by, among others, the x86 and 68k processor families.

2. Subtract with carry. In this convention, the carry flag is set when computing $a - b$ if, and only if, $a \geq b$. This convention is used by, among others, ARM and PowerPC.

The RUN2223 CPU uses the subtract with carry convention[1], but Digital uses the subtract with borrow convention. As a result, you can not use the carry output of the `Subtract` component directly. There are two recommended solutions to implement the `SUB` instruction:

1. Use the `Subtract` component, but invert the carry output.

2. Use the `Adder` component, but invert the b input, and set the carry input to one. This will effectively compute $a - b$ as $a + \text{not}(b) + 1$.

[1]This allows us to translate `SUBf A, c10, D` as `ADDf -c10, A, D` as long as `c10` $\neq 0$, which is needed to implement `CMPf A, c10`.

While performing a 32-bit subtraction, the overflow flag is used in the context of signed operands. It is set if, and only if, the result does not fit into 32-bits. There are exactly two cases where this can occur, determined by looking at the sign of the operands and the result:

Positive - Negative = Negative
Negative - Positive = Positive

Neither the `Adder` nor the `Subtract` components have an overflow output, meaning you will have to implement this logic yourselves.

Bitwise shifts and rotates

While performing a bit shift or rotate operation, the carry flag contains the *last* bit that was shifted/rotated out:

- If a shift/rotate distance of 0 is given, the carry flag is not set.
- If a shift distance of 33 or more is given, the carry flag is not set[2].
- If a shift distance of n is given such that $0 < n \leq 32$, the carry flag is set to bit $32 - n$ of the value to be shifted for a left shift, and bit $n - 1$ for a right shift.
- If a rotate left distance of 1 or more is given, the carry flag is set to bit 0 of the *rotated* value.

Implementing this in Digital is tricky, as none of the components have this exact behavior. As such, we provide sub circuits inside the `ALU` with this exact behavior.

The overflow flag is used to detect whether the sign of the value has changed after shifting/rotating by *one* bit position. As such, it only makes sense to inspect this flag when shifting/rotating by a distance of 1, but we nevertheless define the flag for an arbitrary bit rotation:

- Logical shift left: The overflow flag is set if the sign bit of the *shifted* value differs from the carry flag produced by the shift[3].
- Logical shift right: The overflow flag is set if the sign bit of the value to be shifted is set.
- Rotate left: The overflow flag is set if the sign bit of the *rotated* value differs from the carry flag produced by the rotate.

You have to implement this logical yourselves.

[2]As this ensures a 0 bit is shifted out.

[3]With a shift distance of 1 this effectively means that the sign bit of the value *before* shifting differs from the sign bit of the value *after* shifting, as the carry flag represents the last bit shifted out, i.e. the sign bit of the value before shifting.