Wiktor B. Daszczuk

Processor simulator for the Computer Architecture course

**Faculty of Mathematics and Computer Science**

Task 10 Mon. “Modification of study programs in the fields of study conducted by the Faculty of Mathematics and Computer Science”, implemented as part of the project

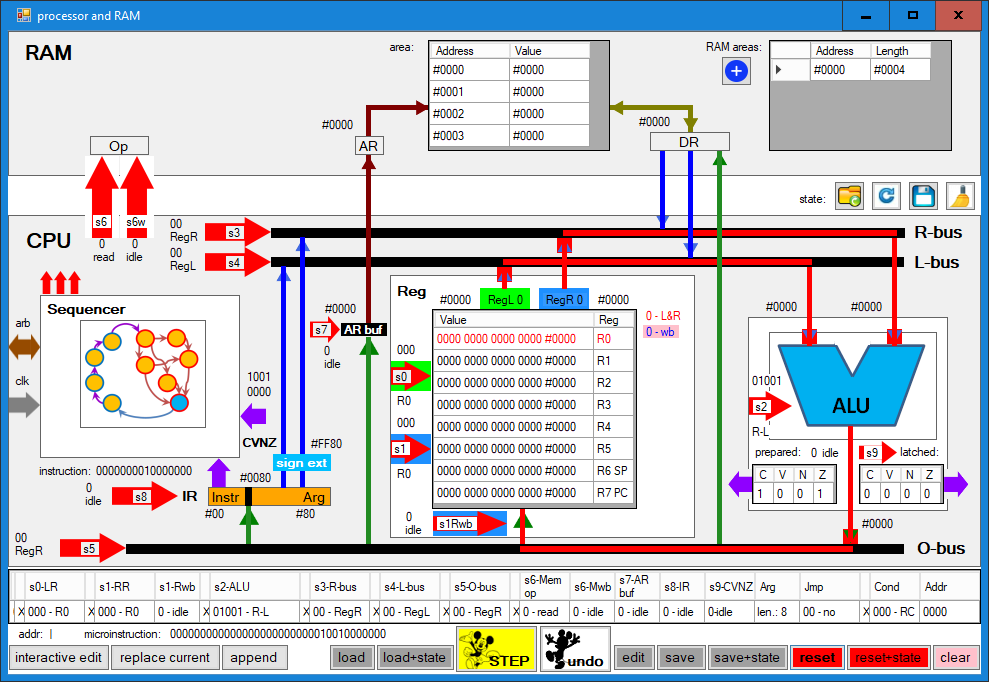
"NERV 2 PW. Science - Education - Development - Cooperation", co-financed by the European Union under the European Social Fund.

The project “NERW 2 PW. Science – Education – Development – Cooperation” is co-financed by the European Union under the European Social Fund

# General description of the simulator

The simulator is built on the level of Register Level Transfer (RTL). Inspired by the PDP 11 architecture, but modernized, especially in terms of microprogramming. The model is 16-bit, both in terms of the length of the addressed memory word, as well as the length of the address and the capacity of the registers.

The general view of the simulator is as follows:



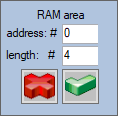
The screen is divided into three fields:

* RAM – primary memory. The simulator sees memory as a non-overlapping list of memory areas, each with a specific base address and length. When a memory reference is made, if the specified address is not in any of the areas, such an area is automatically created.
* CPU - central processor. Three functional blocks can be distinguished in the processor: a set of general purpose registers REG, an arithmetic-logic unit ALU, and a Sequencer. This last block is marked symbolically and there is no insight into how it works, but the user can influence its operation by designing and executing microinstructions.
* Micro-instruction area (at the bottom of the window) – designed or executed micro-instruction consisting of sets of signals controlling elements of individual functional blocks. The microinstruction design and execution offers many options that are activated by the buttons below the microinstruction field.

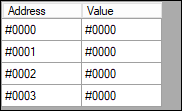
All values in the program, preceded by a # character, are hexadecimal.

# RAM field

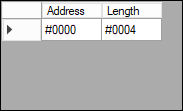
The RAM field offers defining new memory areas after pressing the button. A definition window appears, in which we should enter the base address and area length:



One of the memory areas is always displayed in the "area" table:



The area to be displayed is selected in the table of memory areas on the right by clicking on the row indicated by the arrow:



# CPU

The CPU is the largest field of the simulator, which includes registers, ALU and sequencer. These elements are connected to each other and the RAM area via buses: two buses on the ALU input, L-bus and R-bus, and an output O-bus. The selection of registers, the source/result of individual buses and operations in RAM and ALU are controlled by signal sets coming from microinstructions.

## General Purpose Registers

The register area consists of 8 general purpose registers, numbered from R0 to R7, where R7 is also the program counter PC and R6 is the stack pointer SP. The register value is given in binary (with spaces between nibbles) and in hexadecimal. The value of the register can be modified after clicking on its row, using the window:



As data sources for the ALU, 2 registers, named "green" and "blue", can be selected, the selection of which is marked with the corresponding colors in the simulator. The registers can be the source of data for the ALU (both green and blue registers) or the result (only the blue register). The selection of the green register is controlled by the signal set s0 on a green background, and the selection of the blue register by the set s1 on a blue background. The selection of a register (green or blue) is indicated by the appropriate font color. If a register is both green and blue, it is shown as red. Writing to the blue register is performed by the s1Rwb (write-back) signal. If the write microoperation to the blue register is enabled, it is shown on a pink background.

## ALU

The ALU performs one of the operations on the L and R arguments coming from L-bus and R-bus, respectively:

* 00000 – 0 (constsnt 0)
* 00001 – L (argument from L-Bus)
* 00010 – R (argument from R-Bus)
* 00011 – L-1
* 00100 – R+1
* 00101 – -L
* 00110 – -R
* 00111 – L+R
* 01000 – L-R
* 01001 – R-L
* 01010 – ~L (logcal bitwise negation of L)
* 01011 – ~R (logcal bitwise negation of R)
* 01100 – L|R (bitwise alternative)
* 01101 – L&R (bitwise conjuction)
* 01110 – L xor R (exclusive or)
* 01111 – ~(L xor R)
* 10000 – L<<R.l (L logical shift left by R bits)
* 10001 – L>>R.l (L logical shift right by R bits)
* 10010 – L<<R.a (L arithmetic shift left by R bits)
* 10011 – L>>R.a (L arithmetic shift right by R bits)
* 10100 – L<<R.c (L cyclic shift left by R bits)
* 10101 – L>>R.c (L cyclic shift right by R bits)
* 10110 – L<<R.cc (L cyclic shift left through Carry by R bits)
* 10111 – L>>R.cc (L cyclic shift right through Carry by R bits)

Shifts: l means logical, a - arithmetic, c - cyclic, cc - cyclic through condition register C. << means shift to the left and >> to the right. The operation is selected with the set s2.

Arithmetic operations modify the CVNZ condition bits, shift - C, and arithmetic shift left - also bit V. The condition bits can be latched to the persistent condition register by control signal s9.

## Sequencer

Two elements of the sequencer are visible to the user: the microinstruction (or the entire microprogram) and the instruction register IR. Loading IR from the O-bus bus is performed with the s8 signal. The content of the IR is divided into two parts - the more significant is the opcode and other fields like register number, the less significant is the direct argument. The dividing point (number of bits of the direct operand) is determined by the Arg set. The direct argument can be directed to the L-bus or R-bus, on the way it passes through the sign extension circuit, which extends the argument to 16 bits, where the argument sign bit is replicated (the extension is performed according to the rules in the U2 code, for example for 8-bit operand #80, the extended value is #FF80). The value of the Arg set, which means the length of the direct argument, can be set by clicking the vertical line

the IR , in the window:



The content of the IR can be modified by clicking in its field in the window:



Modifying microinstruction fields will be discussed separately.

## L-bus and R-bus

ALU input buses: L-bus and R-bus can fetch data from registers (L-bus from green and R-bus from blue), direct operand from instruction register IR or memory data register DR. Data exchange with memory will be discussed separately. The L-bus is controlled by the s4 set and the R-bus by the s3 set. The values of s3 and s4 mean:

* 00 – register (green or blue)
* 01 – memory data register DR
* 10 – direct operand from IR
* 11 – N/A

## O-bus

The ALU output bus O-bus directs data to the blue register, the memory data register DR, the instruction register IR or the address buffer ARbuf (cooperation with the memory will be discussed separately). Writing to the blue, IR or ARbuf registers also requires the appropriate write-back control signal: s1Rwb, s8 or s7, respectively. s5 values mean:

* 00 – blue register
* 01 – address buffer ARbuf
* 10 – memory data register DR
* 11 – instruction register IR

## Collaboration with memory

Cooperation with the memory is carried out by control signals s6 (read/write selection), s6w (execute signal for read or write operations), memory address register AR (loads the content automatically from the address buffer ARbuf when the operation is performed) and memory data register DR. When reading, the contents of the memory cell are directed to the DR, and when writing, the content of the DR is directed to the memory cell.

Writing to the memory is done by first setting the address buffer ARbuf and the data register DR, and then setting the signals s6=1 and s6w=1. Reading from the memory is done by first setting the address buffer ARbuf, and then setting the signals s6=0 and s6w=1.

## Summary of control signals

* s0 – 3 bits – green register choice
* s1 – 3 bits – blue register choice
* s1Rwb – 1 bit – blue register write-back
* s2 – 5 bits – ALU microoperation choice
* s3 – 2 bits – source choice for R-bus (blue register, DR, direct argument)
* s4 – 2 bits - source choice for L-bus (green register, DR, direct argument)
* s5 – 2 bits – target choice for O-bus (blue register, ARbuf, DR, IR)
* s6 – 1 bit – memory microoperation: 0-read, 1-write
* s6w – 1 bit – execution of memory microoperation (kind of microoperation selected by s6)
* s7 – 1 bit – write-back to ARbuf
* s8 – 1 bit – write-back to
* s9 – 1 bit – latching temporary Condition bits („prepared”) in persistent Condition bits (“latched”)
* Arg – direct operand size in IR (5 bits, value from 0 to 16)

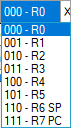
In addition, there are fields in the microinstruction associated with microprogram mirojumps, which will be discussed separately.

# Microinstruction

At the bottom of the window is the currently designed or executing microinstruction:



The microinstruction includes all sets from s0 to Arg, and fields related to microjumps in the microprogram, which will be discussed separately. We can change the value of individual fields by clicking on them - then a list of values for a given bundle appears, for example s0:



Individual fields can also be modified by double-clicking on the arrowheads describing the set names in the CPU area. For multi-bit fields, a list of values appears, for example s3:



while for one-bit packets, the signal is simply negated, for example, for s6w, the memory operation bit is turned on/off (the active state is signaled in red font):



The result of the ALU operation is routed via the O-bus to one of 4 targets: blue register, IR, ARbuf or DR, and requires an appropriate write signal (except DR). These signals can be toggled by double-clicking on the arrowhead ending the branch of the O-bus, for example, for s1Rwb before and after clicking (we get the same as after double-clicking the s1Rwb arrowhead):

In addition, individual sources for L-bus/R-bus or target for O-bus can be selected by double-clicking on the stem of the appropriate arrowheads, for example directing the green register to L-bus, shown before and after clicking:

Note that writing to the registers (except DR) requires directing the O-bus target and the appropriate write-back signal; below, the writing to the blue register is shown in 4 situations: idle, after directing the O-bus to the register, after turning on the write-back, and after turn on both. Only in the latter situation the result of the operation in the ALU will be written to the blue register.

# CPU state

The CPU state includes all registers:

* all memory areas
* address buffer ARbuf
* memory data register DR
* general purpose registers
* instruction register IR
* prepared condition bits CVNZ and persistent bits CVNZ

The CPU state can be written to a file or read from a file. Four buttons on the right below the RAM area denote:

* loading the state from file
* reloading the last loaded state
* saving the state to a file
* state clearing (all registers receive the value 0)

# Microinstruction execution

The execution of microinstructions is triggered by a button



As a result of the microinstruction execution, the control signals will cause corresponding changes in the state of the CPU, in particular:

* passing the appropriate sources for the L and R ALU arguments, through the L-bus and R-bus buses
* execution of the microoperation according to s2 by the ALU
* writing the result from the ALU to the blue register if s5=00 and s1Rwb=1
* writing the result from ALU to ARbuf if s5=01 and s7=1
* writing the result from ALU to DR if s5=10
* writing the result from ALU to IR if s5=11 and s8=1
* writing from the DR register to a memory sell according to ARbuf if s6=1 and s6w=1
* reading from memory to DR according to ARbuf if s6=0 and s6w=1
* writing the temporary CVNZ condition bits to the persistent CVNZ register if s9=1

Microinstruction execution can be undone by pressing a button (restore to previous state).

A greyed out button means that revert to a previous state is not available .

In the simulator window, we can design or execute a single microinstruction.

# Microprogram and microjumps

The microprogram is a sequence of instructions with four-bit microaddresses always numbered from 0000. The microprogram has a maximum of 15 microinstructions, the bit combination 1111 means the end of the microprogram. The microprogram can be written to (and loaded from) an XML file. As part of the microprogram, microjumps can be performed, which is controlled by the Jmp set:

* 00 – no microjump
* 01 – microjump on condition bit 1 (bit choice in Cond set)
* 10 – microjump on condition bit 0 (bit choice in Cond set)
* 11 – unconditional microjump

The Cond set selects the condition bit:

* 000 – bit RC – bit C of latched Condition register
* 001 – bit RV – bit V of latched Condition register
* 010 – bit RN – bit N of latched Condition register
* 011 – bit RZ – bit Z of latched Condition register
* 100 – bit C of prepared Condition register
* 101 – bit V of prepared Condition register
* 110 – bit N of prepared Condition register)
* 111 – bit Z of prepared Condition register

The bits of the operation result conditions in the ALU (prepared) can be used immediately by the sequencer (for branches in the microprogram) and to be rewritten to the "latched" Condition register. The bits of the latched Condition register are part of the processor program model and can be used, for example, in conditional instructions.

The last set is Addr - that is the microaddress to which the microjump is made. The distinguished value 1111 is not a microaddress but the end of the microprogram execution.

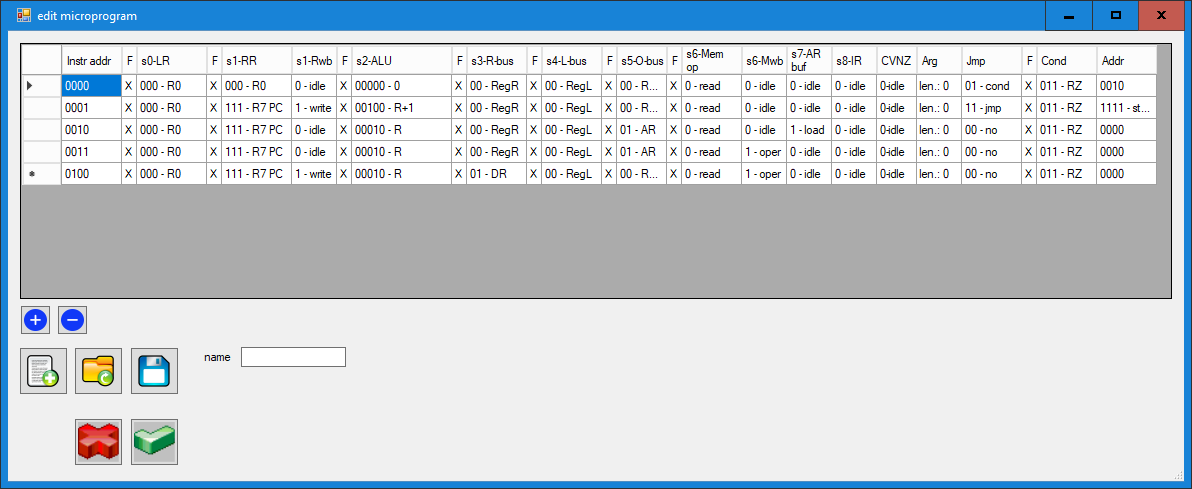
To sum up, the following sets are used to perform microjumps in the microprogram:

* Jmp – 2 bits – no microjump or type of microjump (conditional on 1, conditional on 0, unconditional)
* Cond – selection of condition bit, from ALU result (prepared) or CVNZ latched register
* Addr – microjump destination microaddress (1111 - end of microprogram)

The microinstruction that terminates the microprogram has the following microjump fields:, and a conditional microjump to microaddress 0010 depending on the Z bit of the condition register ("latched") has the fields as follows:.

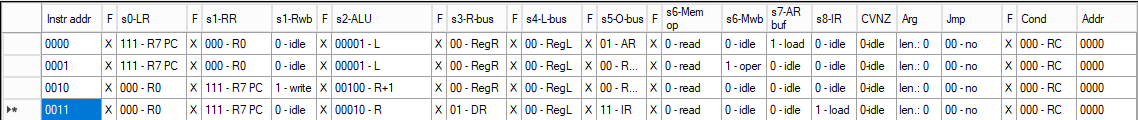
# Editing the microprogram in the modal window

The microprogram can be edited in the design modal window, where we can design the form of the microinstruction, save the microprogram to a file, and read from a file.



The microprogram can be loaded into the sequencer with the accept button. The sequencer microinstruction counter is then set to 0000.

Editing can also be done interactively by invoking the edit window in non-modal mode and sending the projected instruction at the bottom of the simulator window to the microprogram. You can replace the current microprogram instruction (indicated by the arrow) or append it to the end. A typical procedure for designing a microprogram is to design a microinstruction in the simulator, test it (with the possibility of going back in case of an error with the button) and send it to the design window. An exemplary microprogram of the phase of fetching instructions from memory to the IR instruction register, with incrementing of the PC instruction counter, is shown below:



# Interactive editing and microprogram execution

The buttons for editing and executing the microprogram are located at the very bottom of the simulator window. The three buttons on the left bring up the non-modal editor window , send the designed microinstruction to the current place in the design window  or append it to the end .

The button  is used to load the microprogram from a file, and the button  is used to load the microprogram and the CPU state at the same time, the name of the state file must correspond to the name of the microprogram file preceded by the "s\_" prefix, for example, for the microprogram xxx.XML the state file should be named s\_xxx.XML. We can load a state file with a different name using the button .

The STEP  and undo  uttons are used to execute and undo microinstructions, but you can only go back one microinstruction. After retracting the microinstruction, the undo button becomes inactive .

The button  calls up a modal edit window,  saves the current microprogram to a file, and  saves both the microprogram and the CPU state (in separate files, according to the naming given above). The button  returns to the beginning of the microprogram, and  returns both the microprogram and the CPU state to the initial situation. The button  resets the CPU state and microinstruction.

# Horizontal microprogramming

Horizontal microprogramming consists in directing fragments of instruction code directly to sets of CPU control signals. A typical example is the use of a register number in the instruction - then we need to design where the register number is located in the instruction and pass these bits directly to the s0 or s1 set. In order to use horizontal microprogramming, in the appropriate field of the microinstruction, instead of a constant value, use the bits of the instruction starting from the indicated bit. The field that can be microprogrammed is preceded in the microinstruction by a narrow field in which the character x:. In the microprogram edit window, this field looks like this: .

Clicking on this field brings up a window where you can disable the field setting  ->. After disabling this field, it changes its appearance, in which the combination of bits is replaced with the bit number of bit from which the field in the instruction begins and the value of these bits: . After clicking in this field, an instruction field position window appears: , and after accepting, we can see the form of the appropriate bits in the instruction: . In the microprogram edit window, of course, there is no specific instruction, so the appropriate number of x: .

Signal sets that can be horizontally microprogrammed are:

* s0 – 3 bits – green register choice
* s1 – 3 bits – blue register choice
* s2 – 5 bits –ALU microoperation choice
* s3 – 2 bits – choice of source for R-bus (blue register, DR, direct argument)
* s4 – 2 bits – choice of source for L-bus (green register, DR, direct argument)
* s5 – 2 bits – choice of target for O-bus (blue register, ARbuf, DR, IR)
* s6 – 1 bit – memory microoperation: 0-read, 1-write
* Cond – selection of the condition bit to microjump in the microprogram, from the result of the ALU (prepared) or the persistent CVNZ register

Below is an example of a microinstruction that adds the blue and green registers and places the result in blue register, where the blue register number is in bits 9,10,11 of the instruction and the green register number is in bits 6,7,8.



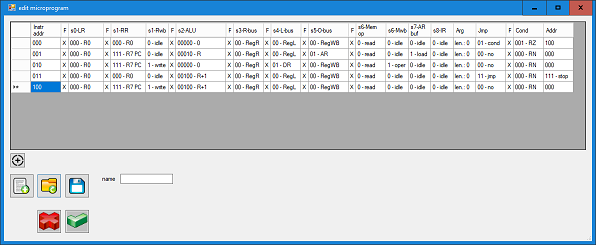
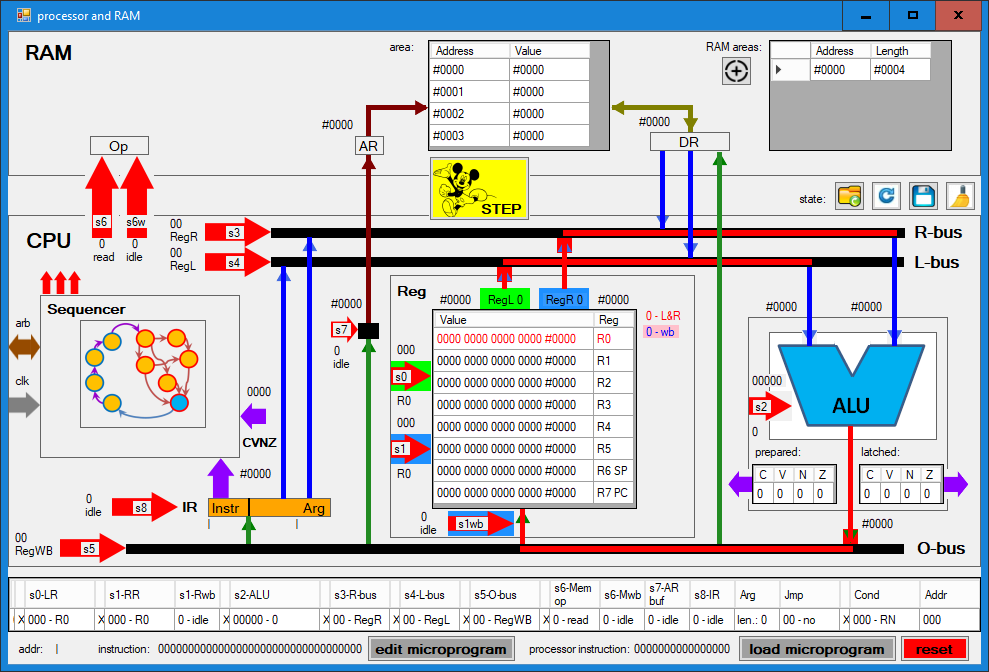
Enlarged essential part of the microinstruction:



# Simulator in the context of processor architecture

The simulator does not implement the full model of the processor. The diagram below shows the concept of full loading and execution of instructions in a microprogrammed processor. The parts marked in red are missing in the simulator (op-code decoding, calling a microprogram with parameters and microprogram chaining).

**wywołanie**



**instrukcja**

**mikroprogram**

**następny mikroprogram**

**dekodowanie**

In the simulator, we can load and execute a microprogram for fetching instructions from memory to the IR, and then manually load to the simulator and execute a specific executive microprogram. There is no instruction decoder in the simulator that would determine the necessary microoperations to execute the instruction and call the appropriate microprogram: this must be done manually.

There is also no mechanism for chaining microprograms, allowing, for example, in one microprogram to perform calculations in registers, and in a separate microprogram to send the result to a memory cell in accordance with the relative addressing scheme (both can use horizontal microprogramming mechanisms, for example to specify a register number). It is up to the user to load and execute subsequent microprograms.

Automation of instruction decoding and microprogram chaining will be considered in further development of the simulator.