BUDAPEST UNIVERSITY OF TECHNOLOGY AND ECONOMICS
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATICS
DEPARTMENT OF MEASUREMENT AND INFORMATION SYSTEMS

The MiniRISC processor

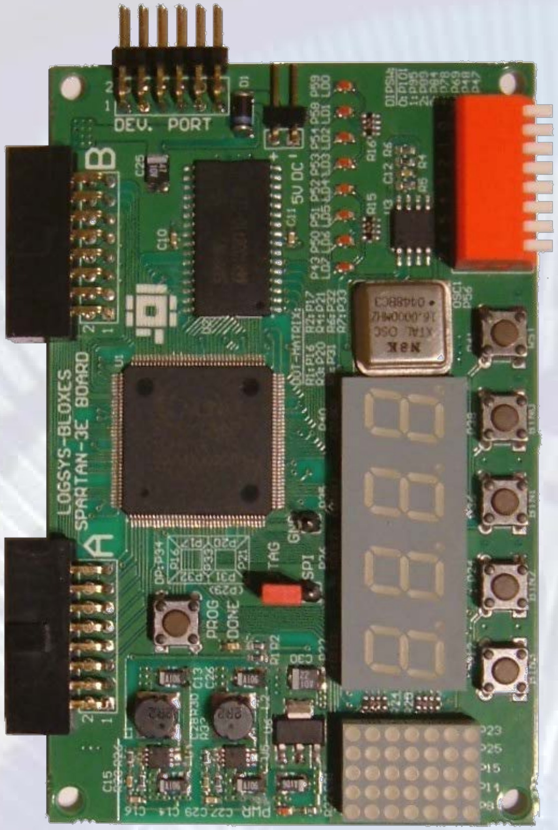
Béla Fehér, Tamás Raikovich, Attila Fejér

BUTE DMIS

Contents

MiniRISC processor - Introduction

- **8-bit microprocessor for simple applications**
- **Fits in well with the complexity of the LOGSYS Spartan-3E FPGA board**
- **Low resource requirement**
- **Harvard architecture**
  - 256 x 16 bit program memory
  - 256 x 8 bit data memory
- **Simple RISC instruction set**
  - Load/store architecture
  - 16 x 8 bit internal register file
  - Operations on register file only



MiniRISC processor - Introduction

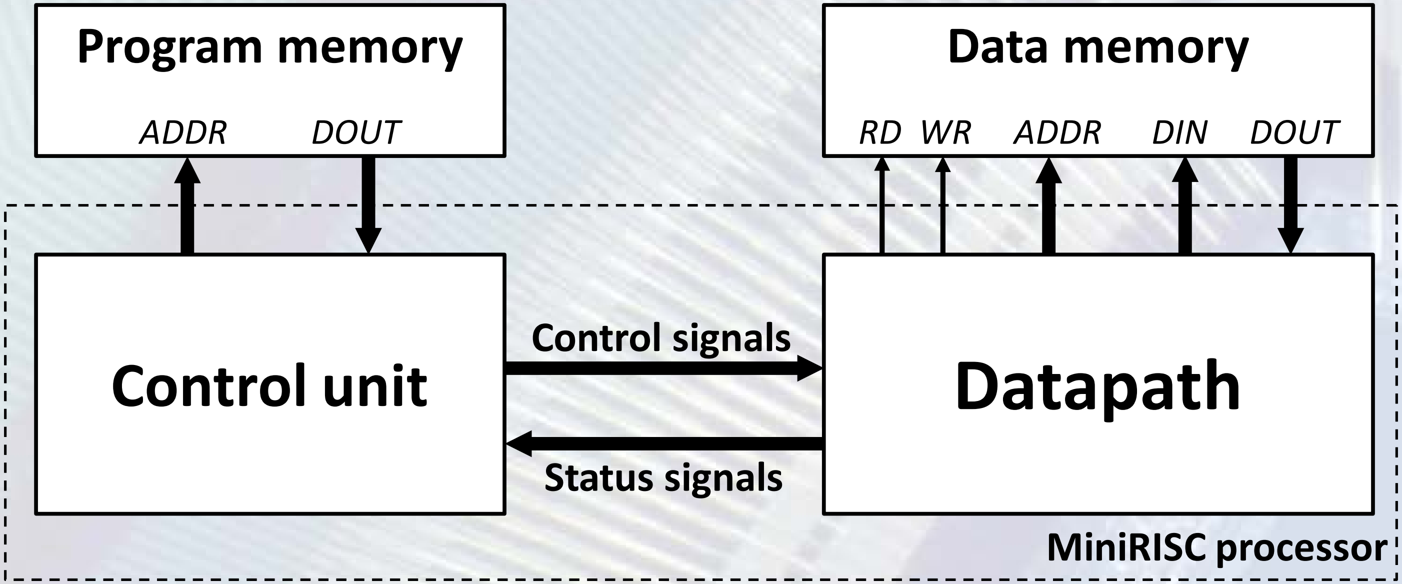
- **Simple RISC instruction set**
  - Data moving instructions
  - Arithmetic instructions (+, -, compare)
  - Logic instructions (AND, OR, XOR, bit test)
  - Shift, rotate and swap instructions
  - Program control instructions
- **Operands: two registers or a register and an 8-bit constant**
- **Absolute and register indirect addressing modes**
- **Zero (Z), carry (C), negative (N), overflow (V) status bits**
  - Conditional jump instructions for testing
- **Jumps can be done to the whole program memory address range**

Contents

MiniRISC processor - Structure

Its internal structure follows the RTL design method:

- ***Control unit:*** **fetching and processing the instructions, and controlling the datapath accordingly**
- ***Datapath:*** **executing the operations on the data**



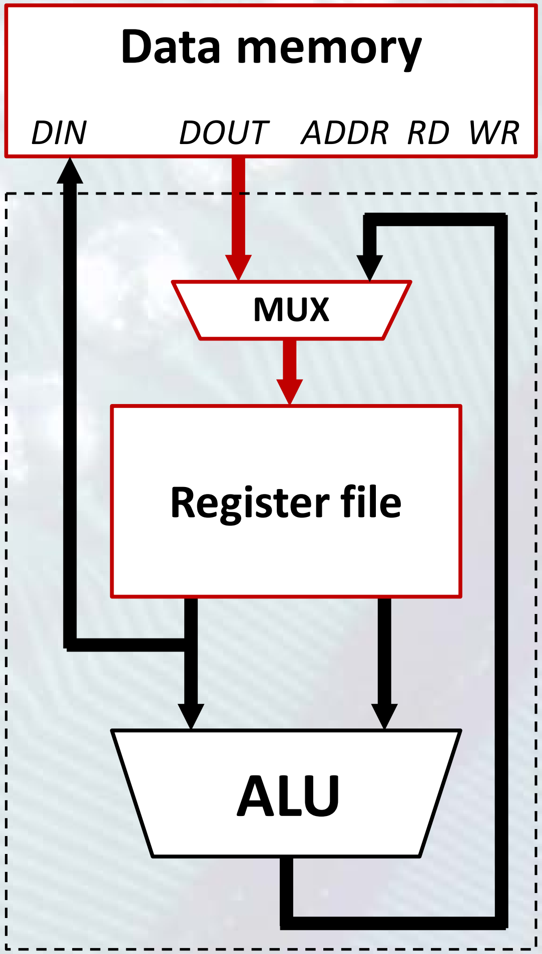
MiniRISC processor - Structure

(Datapath)

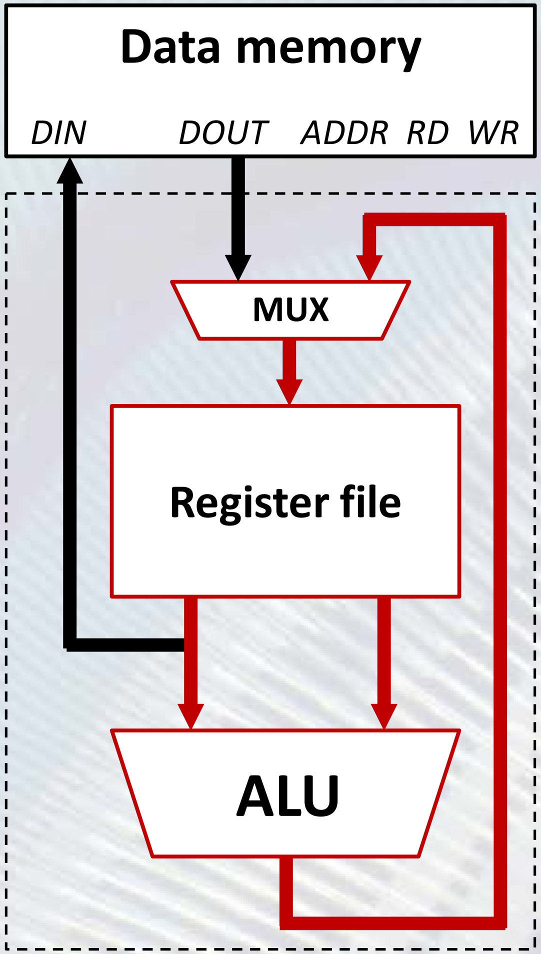
- **The operations are executed on the data in the datapath**
- **Main steps of the data processing:**
  1. Loading the data
  2. Transforming these data
  3. Storing the result
- **The basic datapath therefore:**
  - Reads and writes the external data memory where the main data exists
  - Contains a register file to hold the data locally
  - Contains an arithmetic-logic unit (ALU) to transform the local data

MiniRISC processor - Structure

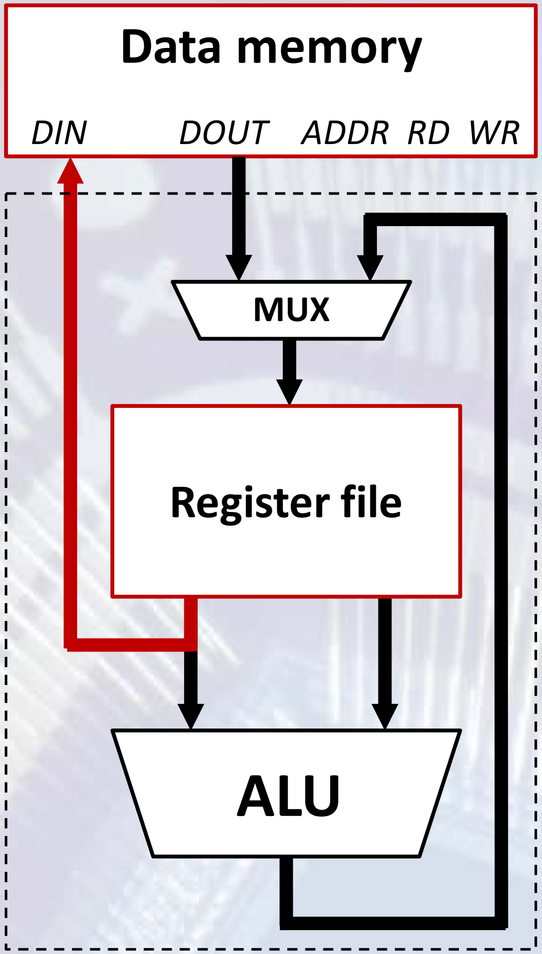
(Datapath)



Data memory read (load)



Transforming the local data (ALU operation)

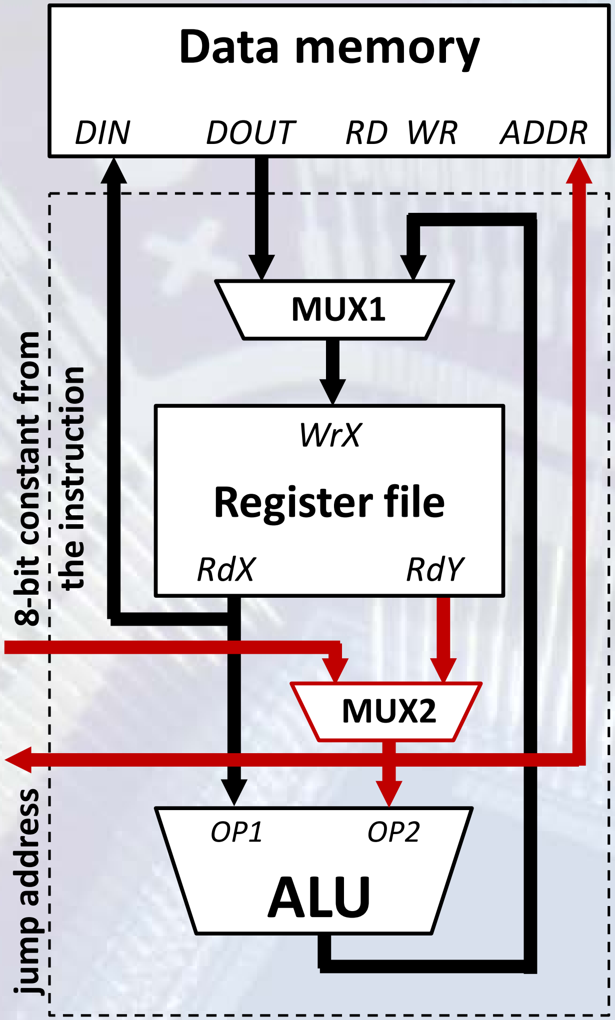


Data memory write (store)

MiniRISC processor - Structure

(Datapath of the MiniRISC processor)

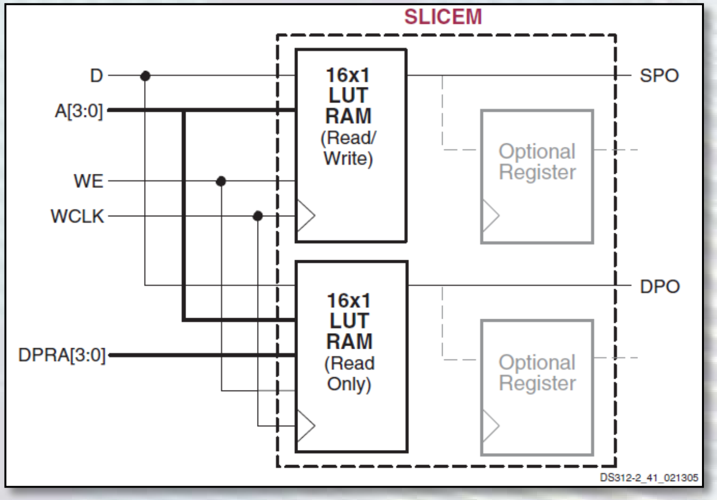
- **Extended version of the basic datapath**
- **Write data select multiplexer for the register file (MUX1)**
  - ALU result or data memory
- **16 x 8 bit register file**
  - Two register addresses are used
- **Arithmetic-logic unit (ALU)**
- **Selecting the 2nd ALU operand (MUX2)**
  - Register
  - 8-bit constant from the instruction
- **Selecting the addressing mode (MUX2)**
  - Absolute: address is from the instruction
  - Indirect: address is from the register file

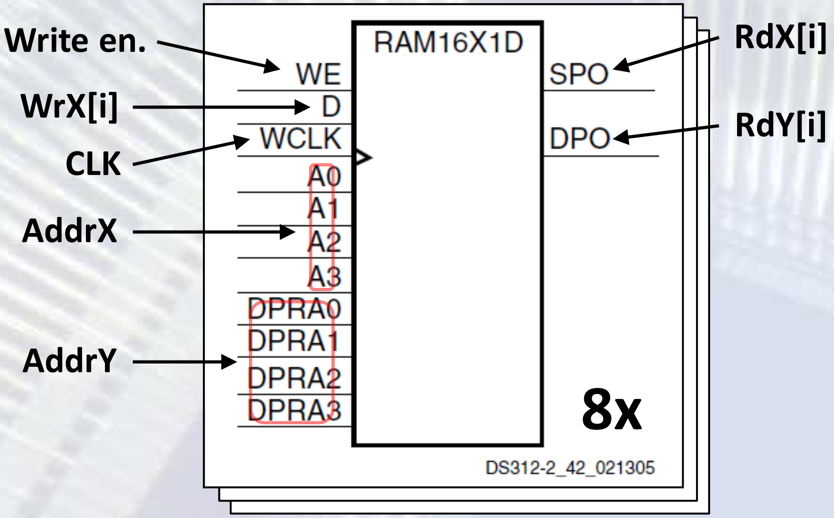


MiniRISC processor - Structure

(Datapath of the MiniRISC processor – Register file)

- **The register file is implemented using distributed RAM (FPGA resource)**
- **Distributed RAM can be used to store small amount of data efficiently**
  - 1 write port and 1 or 2 read ports
    - The write and the first read port has shared address input (**A=AddrX**)
    - The address for the second read port can be different (**DPRA=AddrY**)
  - The write operation is synchronous
    - Happens after the clock event (rising or falling edge) if enabled (**WE=1**)
  - The read operation is asynchronous
    - The addressed data appears ”immediately” on the data output

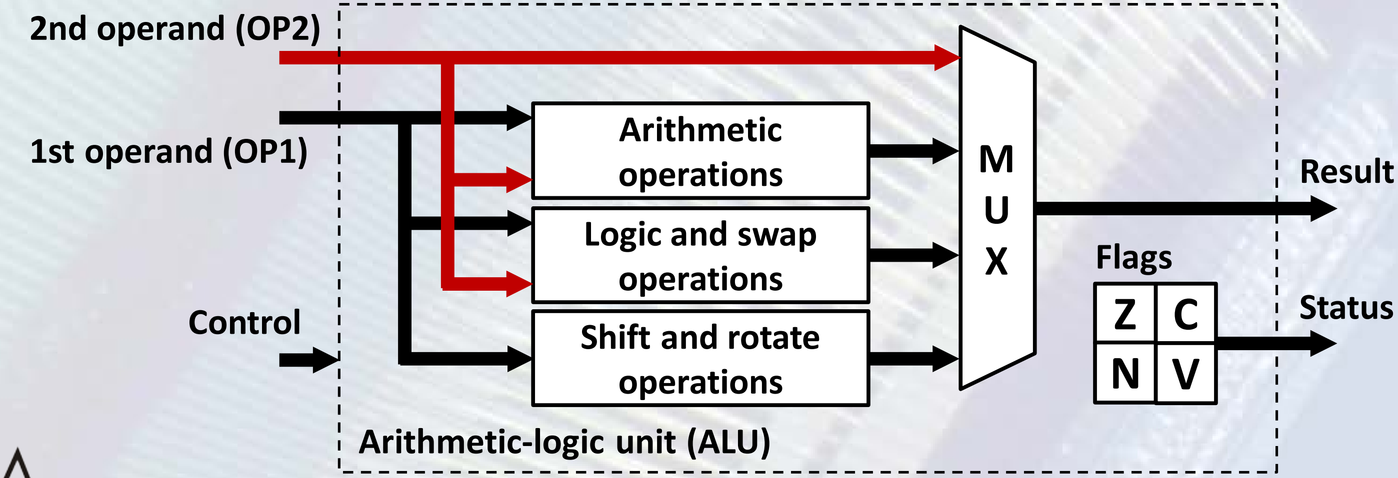




MiniRISC processor - Structure

(Datapath of the MiniRISC processor – ALU)

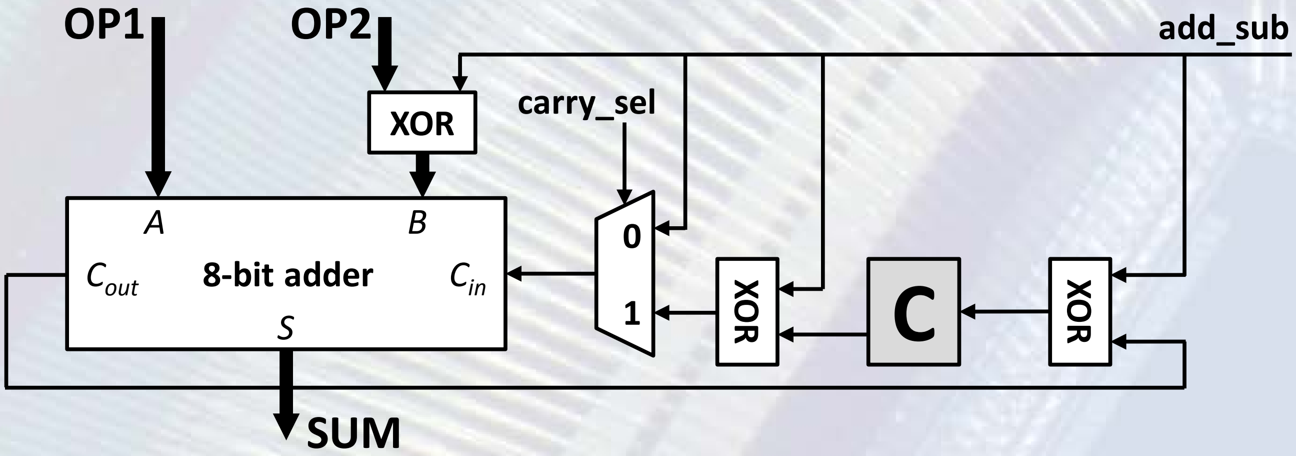
- **The ALU executes different operations on the local data**
  - Data moving: no operation, the result is OP2
  - Arithmetic: addition and subtraction with or without carry
  - Logic: bitwise AND, OR, XOR
  - Shift, rotate and swap
- **Status flags give information about the result of the operations**
  - Zero (Z), carry (C), negative (N) and overflow (V) status flags
  - Their value can be tested using the conditional jump instructions



MiniRISC processor - Structure

(Datapath of the MiniRISC processor – ALU)

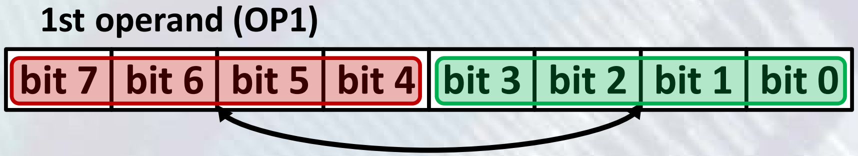
- **Arithmetic operations**
  - In case of the Xilinx FPGAs, the adder/subtractor circuit doesn’t have carry input (C_{in}), therefore the arithmetic operations are implemented by the following way:
    - Addition without carry: $\{C_{out}, SUM\} = OP1 + OP2 + 0$
    - Addition with carry: $\{C_{out}, SUM\} = OP1 + OP2 + C_{in}$
    - Subtraction without borrow: $\{\overline{C_{out}}, SUM\} = OP1 + \overline{OP2} + 1$
    - Subtraction with borrow: $\{\overline{C_{out}}, SUM\} = OP1 + \overline{OP2} + \overline{C_{in}}$



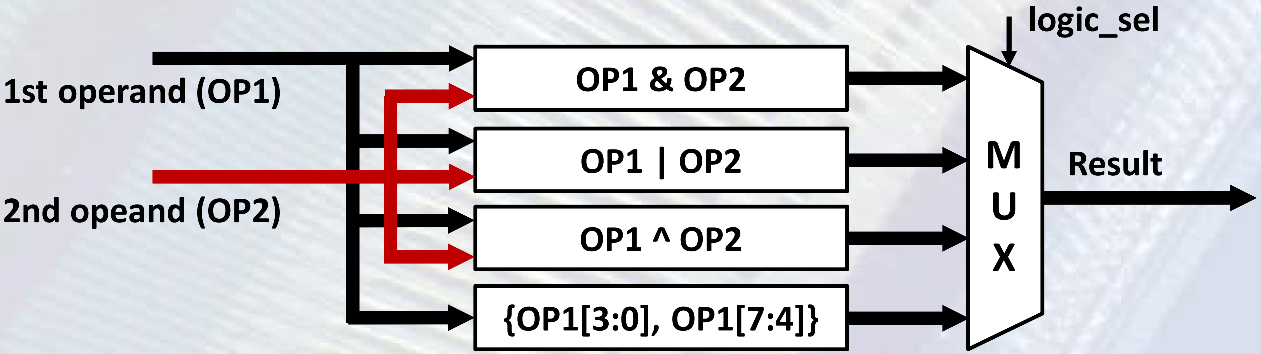
MiniRISC processor - Structure

(Datapath of the MiniRISC processor – ALU)

- **Logic operations**
  - Bitwise AND, OR and XOR operations
- **Swap operation**
  - Swapping the lower and upper 4 bits of the 1st operand



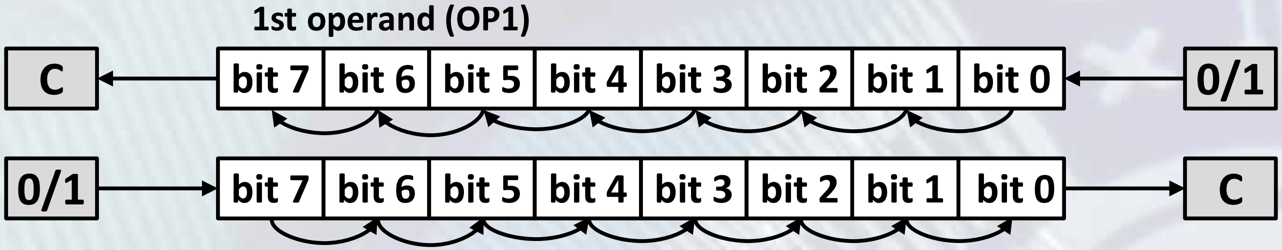
  - Implemented in the logic operation block
    - The 4th input of the MUX can be used for the swap operation
    - Same status flags are modified (Z and N) → same control



MiniRISC processor - Structure

(Datapath of the MiniRISC processor – ALU)

- **Logic shift**
  - The shift direction can be left or right
  - The shifted in bit can be 0 or 1, the shifted out bit is stored in the C flag

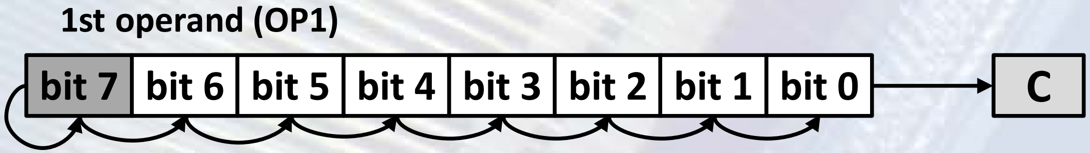


1st operand (OP1)

C ← bit 7 | bit 6 | bit 5 | bit 4 | bit 3 | bit 2 | bit 1 | bit 0 ← 0/1

0/1 → bit 7 | bit 6 | bit 5 | bit 4 | bit 3 | bit 2 | bit 1 | bit 0 → C

- **Arithmetic shift right**
  - When a signed number is shifted right, the value of the sign bit (MSb) should be preserved in order to get correct result
  - The shifted out bit is stored in the carry (C) flag
  - There is no separate arithmetic shift left operation because it is the same as the logical shift left operation



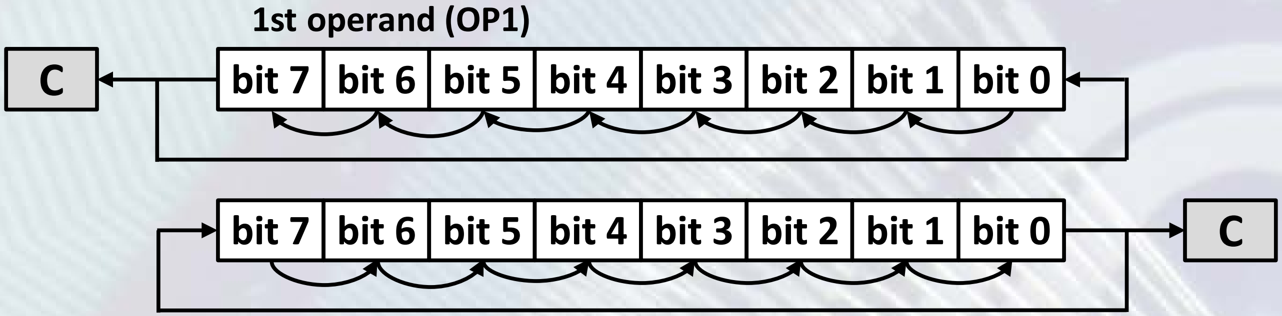
1st operand (OP1)

bit 7 | bit 6 | bit 5 | bit 4 | bit 3 | bit 2 | bit 1 | bit 0 → C

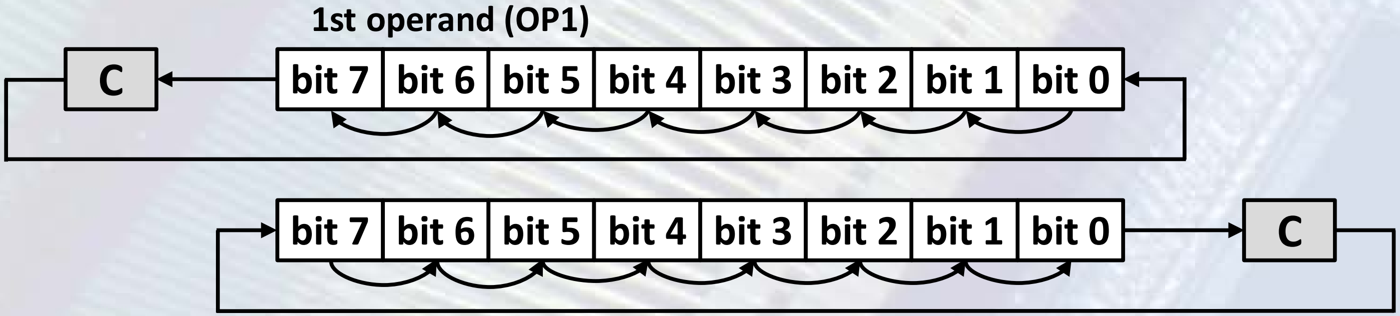
MiniRISC processor - Structure

(Datapath of the MiniRISC processor – ALU)

- **Normal rotate**
  - The rotate direction can be left or right
  - The shifted out bit is shifted in at the other side
  - The shifted out bit is stored in the carry (C) flag



- **Rotate through the carry (C) flag**



MiniRISC processor - Structure

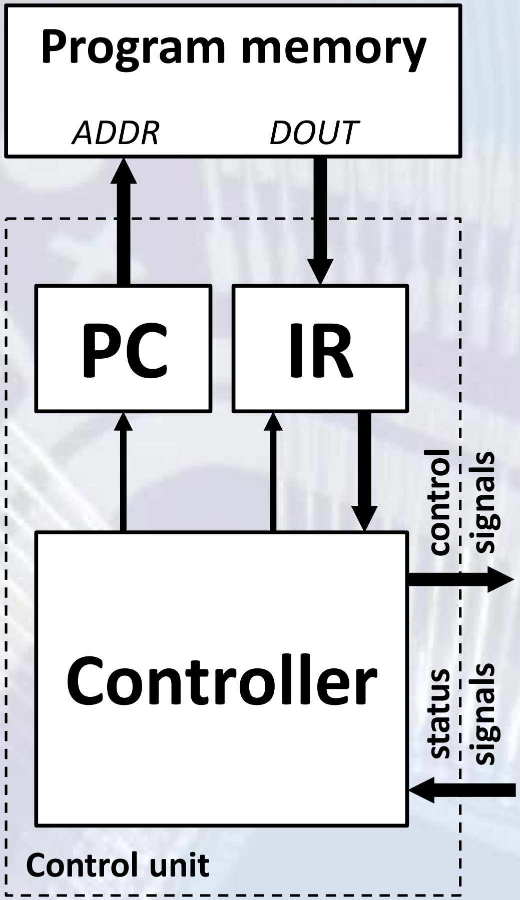
(Datapath of the MiniRISC processor – ALU)

- **The ALU status bits give information about the result of the operations**
- ***Zero bit (Z)***
  - Indicates if the result of the operation is zero
  - The data moving operations don't change the Z flag
- ***Carry bit (C)***
  - Indicates if carry has been generated by the arithmetic operations
  - In case of shift/rotate operations, the shifted out bit is stored here
- ***Negative bit (N)***
  - Two's complement sign bit, the MSb (bit 7) of the result
  - The data moving operations don't change the N flag
- ***Overflow bit (V)***
  - Indicates the two's complement overflow
    - The result of the arithmetic operation cannot be represented using 8 bits
  - Detection: the sign bit of the operands are the same but the sign bit of the result is different (the C_{in7} ***xor*** C_{out7} method cannot be used because the carry in bit of the MSb is not available inside the FPGA)

MiniRISC processor - Structure

(Control unit)

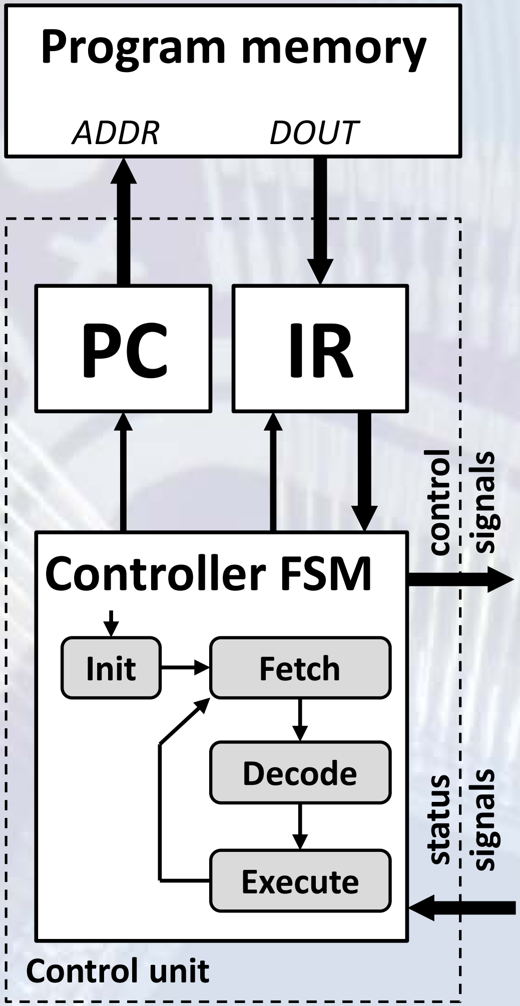
- **Example: DMEM[3] = DMEM[0] + DMEM[1], this requires 4 datapath operations:**
  1. REG[0] = DMEM[0] (load)
  2. REG[1] = DMEM[1] (load)
  3. REG[1] = REG[0] + REG[1] (ALU operation)
  4. DMEM[3] = REG[1] (store)
- **Instruction: operation that the CPU can execute**
- **Program: series of instructions**
  - The given task has to be decomposed into processor-supported instructions
- **The program is stored in the program memory**
- **The control unit reads the instructions and executes them on the datapath**
  - ***Program Counter (PC):*** generates the address of the current instruction
  - ***Instruction Register (IR):*** stores the instruction read from the prg. mem.



MiniRISC processor - Structure

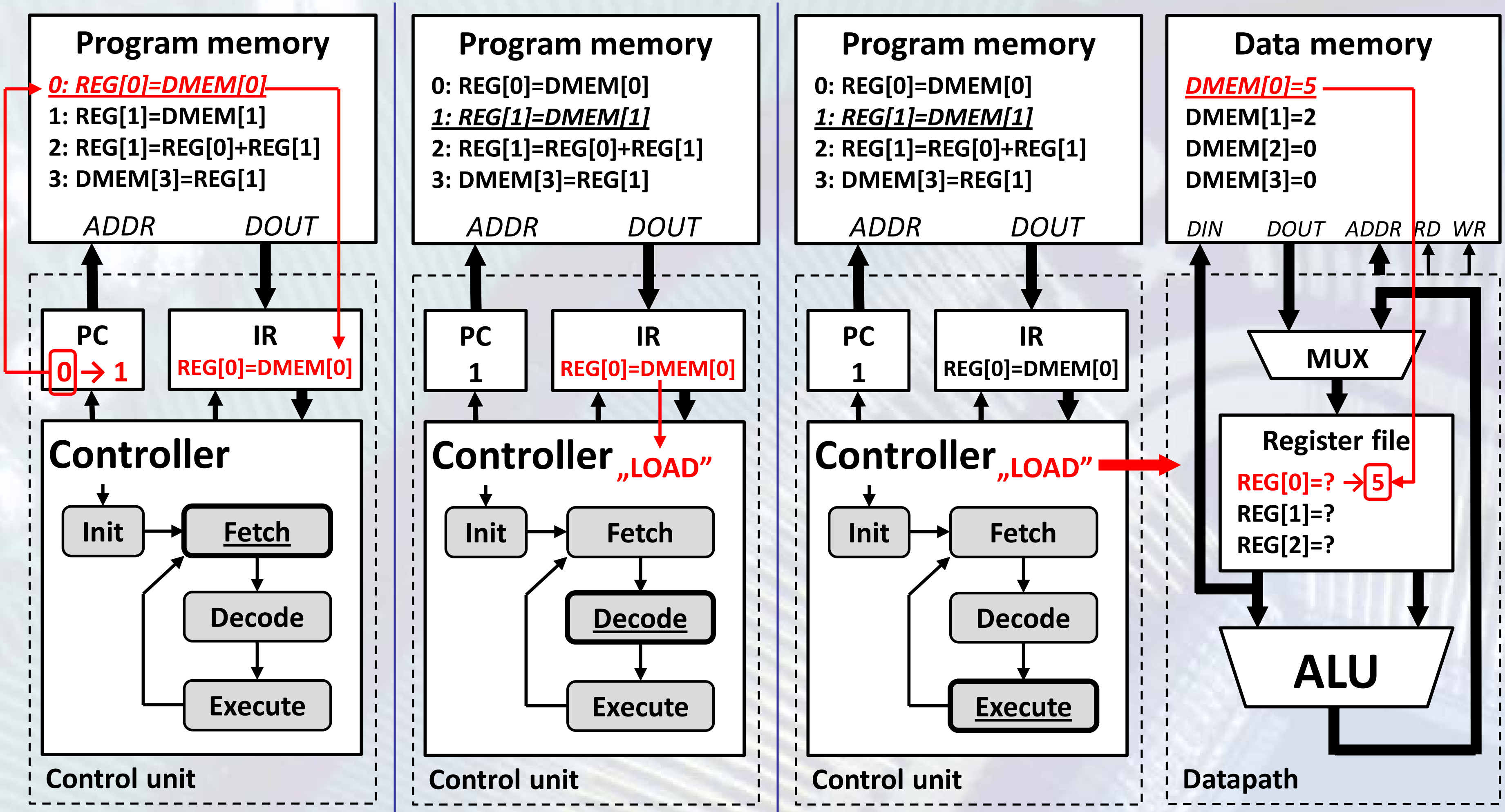
(Control unit)

- **To carry out each instruction, the control unit does the following steps:**
  - ***Fetch:*** reading the instruction from the program memory and incrementing the program counter
  - ***Decode:*** determining the operation and its operands
  - ***Execute:*** carrying out the instruction's operation using the datapath
- **The controller can be an FSM**
  - One state of the controller FSM can be associated to each step above
  - In this case, processing an instruction requires three clock cycles



MiniRISC processor - Structure

(Control unit)



Program memory

0: REG[0]=DMEM[0]
1: REG[1]=DMEM[1]
2: REG[1]=REG[0]+REG[1]
3: DMEM[3]=REG[1]

ADDR *DOUT*

PC 0 → 1

IR REG[0]=DMEM[0]

Controller

Init → **Fetch** → Decode → Execute

Control unit

Instruction fetch

Program memory

0: REG[0]=DMEM[0]
1: REG[1]=DMEM[1]
2: REG[1]=REG[0]+REG[1]
3: DMEM[3]=REG[1]

ADDR *DOUT*

PC 1

IR REG[0]=DMEM[0]

Controller „LOAD”

Init → Fetch → **Decode** → Execute

Control unit

Decode

Program memory

0: REG[0]=DMEM[0]
1: REG[1]=DMEM[1]
2: REG[1]=REG[0]+REG[1]
3: DMEM[3]=REG[1]

ADDR *DOUT*

PC 1

IR REG[0]=DMEM[0]

Controller „LOAD”

Init → Fetch → Decode → **Execute**

Control unit

Data memory

DMEM[0]=5
DMEM[1]=2
DMEM[2]=0
DMEM[3]=0

DIN *DOUT* *ADDR* *RD* *WR*

MUX

Register file

REG[0]=? → 5
REG[1]=?
REG[2]=?

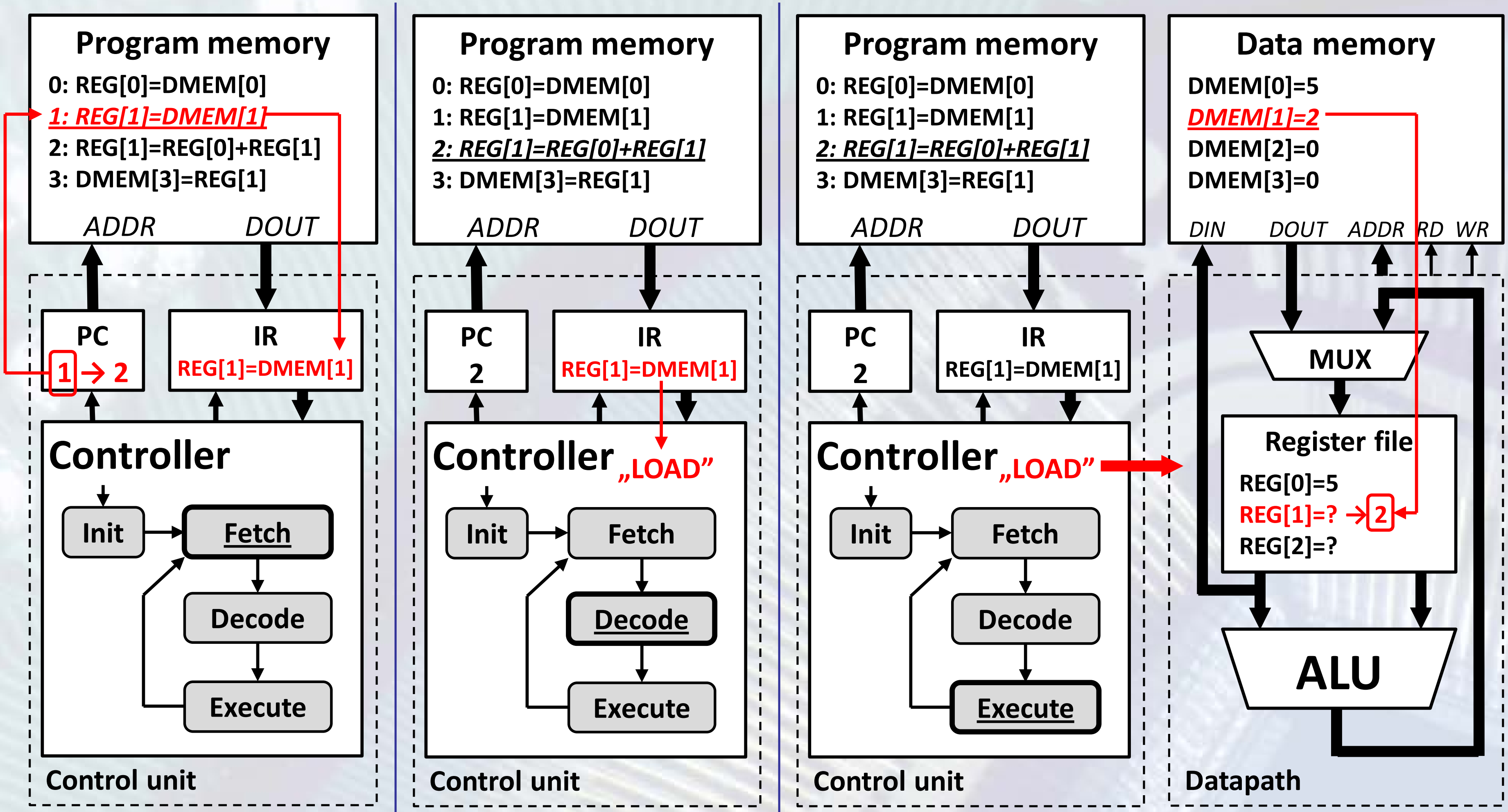
ALU

Datapath

Execute

MiniRISC processor - Structure

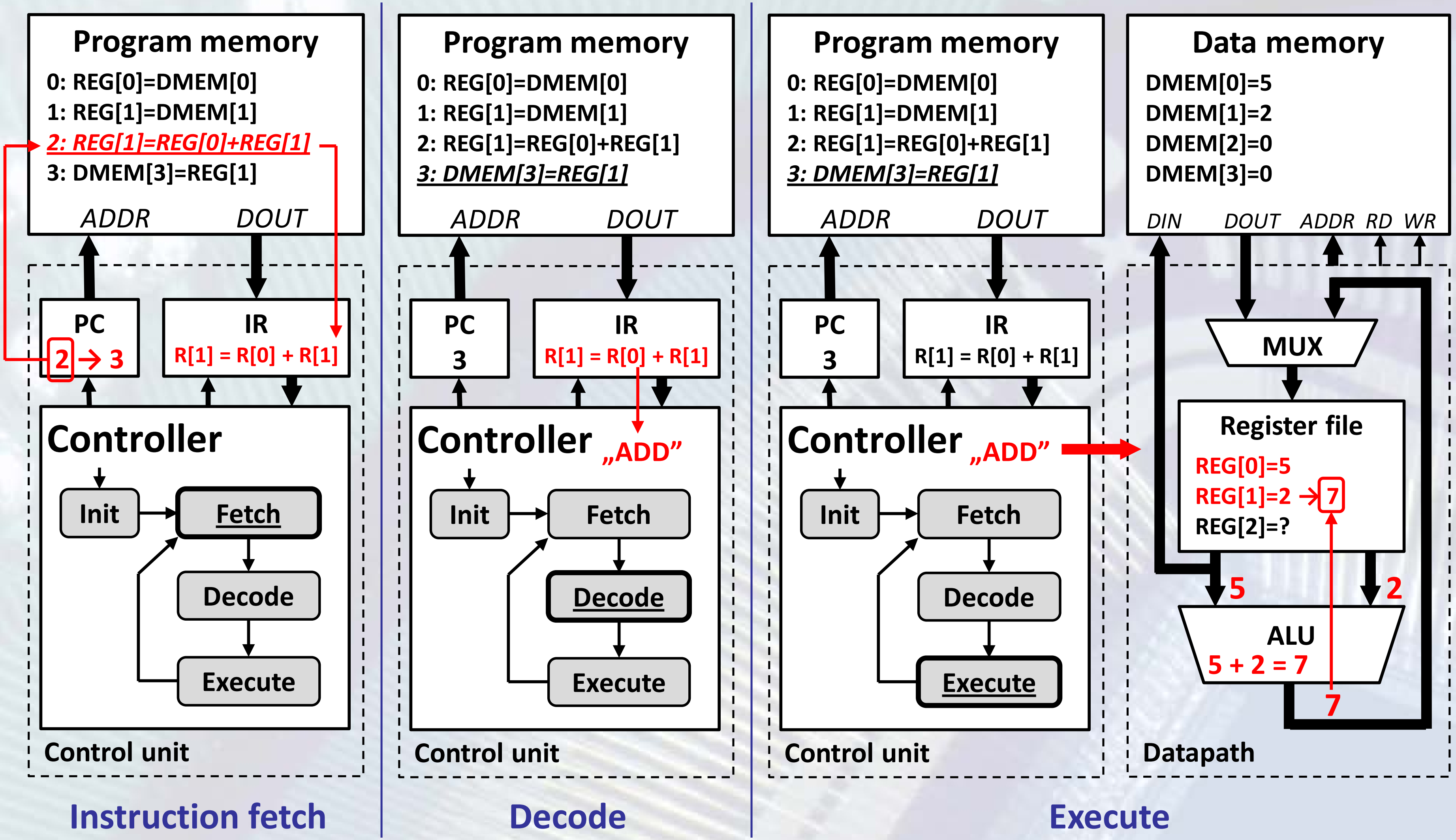
(Control unit)



Instruction fetch **Decode** **Execute**

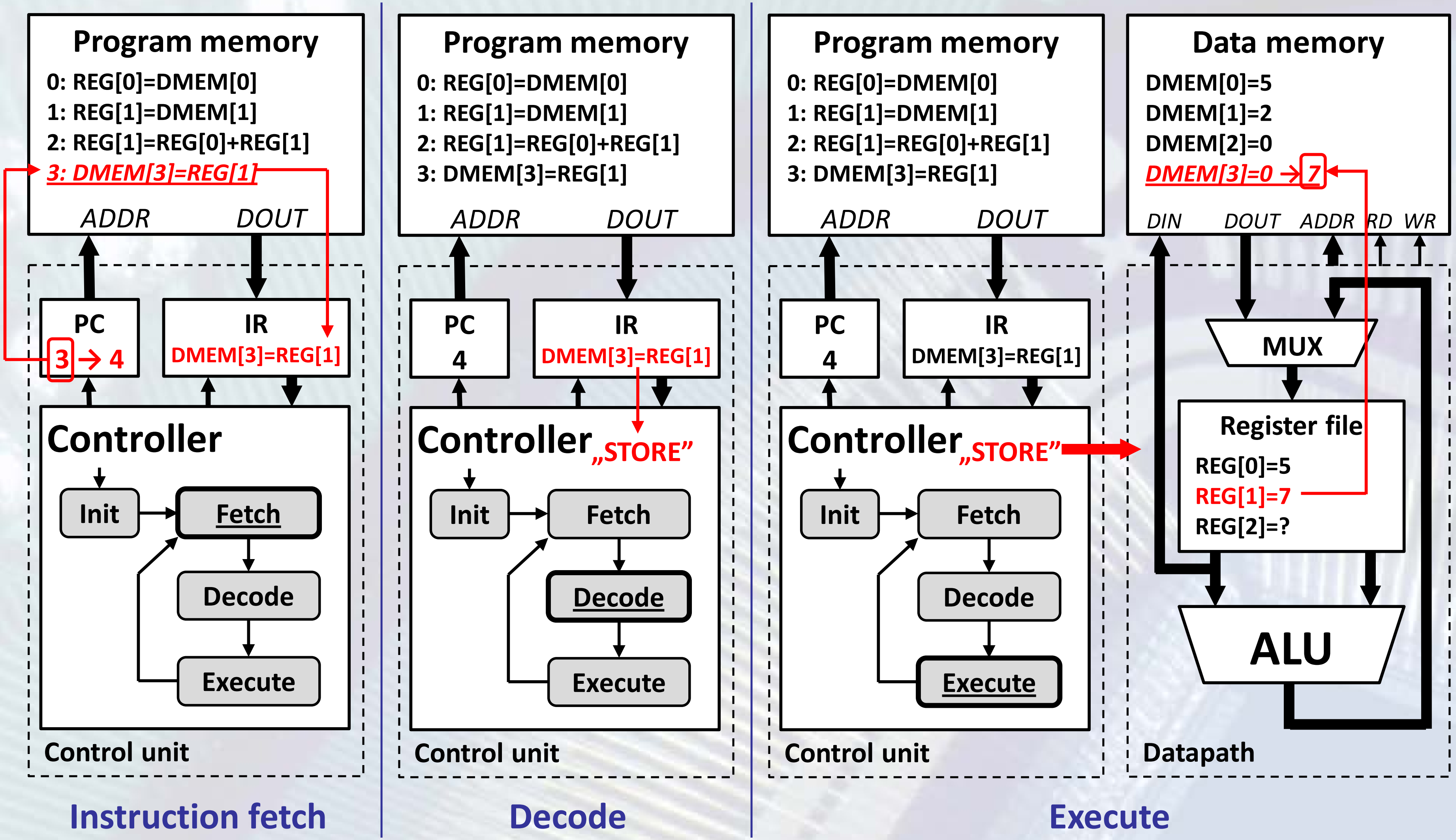
MiniRISC processor - Structure

(Control unit)



MiniRISC processor - Structure

(Control unit)



MiniRISC processor - Structure

(Control unit)

- **The program memory contains the instructions in form of binary code (machine code)**
  - The processor cannot interpret higher-level descriptions
- **Every instruction contains the description of the operation (opcode) and the other required data (operands)**

🟥 operation (4 bits) 🟦 reg. address (4 bits) 🟩 mem. address (8 bits)

Program memory

Addr.	Operation	Machine code (16 bits)	Assembly code
0:	REG[0] = DMEM[0]	1101000000000000	`MOV r0, 0x00`
1:	REG[1] = DMEM[1]	1101000100000001	`MOV r1, 0x01`
2:	REG[1] = REG[0] + REG[1]	1111000100000000	`ADD r1, r0`
3:	DMEM[3] = REG[1]	1001000100000011	`MOV 0x03, r1`

MiniRISC processor - Structure

(Control unit of the MiniRISC CPU – Instruction set)

- **Instruction set: list of the allowed instructions and their representation in memory**
- **Basically, an instruction is a binary number**
  - Machine code
- **Structure of the instructions**
  - ***Opcode:*** determines the operation to be executed
  - ***Operands:*** data used by the given operation
    - Register
    - Constant value

MiniRISC processor - Structure

(Control unit of the MiniRISC CPU – Instruction set)

- **The size of the MiniRISC instructions is 16 bits in order to use the FPGA resources more efficiently (less prg. mem. size)**
- **The register file contains 16 registers**
  - ***Register operand: 4-bit address***
  - 16 register is enough for most of the tasks
  - 16-bit instructions → more registers are not possible
- **8-bit datapath**
  - ***Constant operand: 8-bit value***
  - Operations with constants are very common, therefore the usage of constant operands in case of ALU operations greatly reduces the program size
- **256 word program and data memory → 8-bit mem. address**
  - The whole address range can be covered using absolute or register indirect addressing

MiniRISC processor - Structure

(Control unit of the MiniRISC CPU – Instruction set)

- **How many operands should be in the 16-bit instructions?**
- **Separate addresses for each port of the register file**
  - The program contains less number of instructions
  - The instructions are wider
  - Operands: 12 bits
    - Two source register (rX, rY) and a destination register (rD)
    - One register (rD) and an 8-bit constant
  - Opcode: 4 bits → 16 possibile opcodes
    - ***16 opcodes are not enough for the MiniRISC processor!***

15 12	11 8	7 4	3 0
Opcode	**rD**	**rX**	**rY**

15 12	11 8	7 0
Opcode	**rD**	**8-bit constant**

MiniRISC processor - Structure

(Control unit of the MiniRISC CPU – Instruction set)

- **How many operands should be in the 16-bit instructions?**
- **Two register addresses**
  - Good tradeoff between the number of instructions and the length of instructions
  - Two main instruction type according to the operands
    - A register (rX) and an 8-bit constant → 12 bits A type
    - Two registers (rX and rY, rX is the dest. reg.) → 8 bits B type
  - Opcode: 4 bits + 4 bits
    - A type: 15 opcodes (B type is indicated by the 1111 prefix)
    - B type: 16 opcodes
    - ***31 opcodes together, this is enough for the MiniRISC processor***

A type instruction:

15 12	11 8	7 0
opcode	**rX / control**	**8-bit constant**

B type instruction:

15 12	11 8	7 4	3 0
1111	**rX / control**	**opcode**	**rY / control**

MiniRISC processor - Structure

(Control unit of the MiniRISC CPU – Instruction set)

- **An instruction is a binary number (machine code)**
- **Machine code is hard to work with → assembly code**
- **The assembly code uses mnemonics**
  - Mnemonic: short word, refer to the operation
    - For example: ADD – addition, MOV – data movement, etc.
  - Operands of the MiniRISC instructions can be:
    - Register: r0 – r15
    - Constant: #0 – #255 (for ALU operations)
    - Memory address: 0 – 255 (constant for memory addressing)
    - Register for indirect addressing: (r0) – (r15)
- **The assembler generates the machine code from the assembly code**

MiniRISC processor - Structure

(Control unit of the MiniRISC CPU – Instruction set)

Data moving instructions

- Data memory read using absolute or indirect addressing
- Data memory write using absolute or indirect addressing
- Load constant into register
- Move data from register to register
- The value of the ALU status bits is preserved

Machine code	Assembly code	Operation	Z	C	N	V
1101xxxxaaaaaaaa	MOV rX, addr	rX ← DMEM[addr]	-	-	-	-
1111xxxx1101yyyy	MOV rX, (rY)	rX ← DMEM[rY]	-	-	-	-
1001xxxxaaaaaaaa	MOV addr, rX	DMEM[addr] ← rX	-	-	-	-
1111xxxx1001yyyy	MOV (rY), rX	DMEM[rY] ← rX	-	-	-	-
1100xxxxiiiiiiii	MOV rX, #imm	rX ← imm	-	-	-	-
1111xxxx1100yyyy	MOV rX, rY	rX ← rY	-	-	-	-

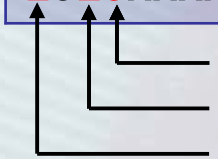
MiniRISC processor - Structure

(Control unit of the MiniRISC CPU – Instruction set)

Arithmetic instructions

- Addition and subtraction with or without carry
- Compare (subtraction without storing the result)
- Operands: two registers or a register and an 8-bit constant
- Some opcode bits are used as datapath control signals
  - This way, the controller state machine becomes simpler

Machine code	Assembly code	Operation	Z	C	N	V
`0000xxxxiiiiiiii`	`ADD rX, #imm`	rX ← rX + imm	+	+	+	+
`0001xxxxiiiiiiii`	`ADC rX, #imm`	rX ← rX + imm + C	+	+	+	+
`0010xxxxiiiiiiii`	`SUB rX, #imm`	rX ← rX – imm	+	+	+	+
`0011xxxxiiiiiiii`	`SBC rX, #imm`	rX ← rX – imm – C	+	+	+	+
`1010xxxxiiiiiiii`	`CMP rX, #imm`	rX – imm	+	+	+	+



Carry select (0: without carry, 1: with carry)

Operation select (0: addition, 1: subtraction)

1 if the result of the arithmetic operation is not stored

MiniRISC processor - Structure

(Control unit of the MiniRISC CPU – Instruction set)

Arithmetic instructions

- Addition and subtraction with or without carry
- Compare (subtraction without storing the result)
- Operands: two registers or a register and an 8-bit constant
- Some opcode bits are used as datapath control signals
  - This way, the controller state machine becomes simpler

Machine code	Assembly code	Operation	Z	C	N	V
`1111xxxx0000yyyy`	`ADD rX, rY`	rX ← rX + rY	+	+	+	+
`1111xxxx0001yyyy`	`ADC rX, rY`	rX ← rX + rY + C	+	+	+	+
`1111xxxx0010yyyy`	`SUB rX, rY`	rX ← rX – rY	+	+	+	+
`1111xxxx0011yyyy`	`SBC rX, rY`	rX ← rX – rY – C	+	+	+	+
`1111xxxx1010yyyy`	`CMP rX, rY`	rX – rY	+	+	+	+

Carry select (0: without carry, 1: with carry)

Operation select (0: addition, 1: subtraction)

1 if the result of the arithmetic operation is not stored

MiniRISC processor - Structure

(Control unit of the MiniRISC CPU – Instruction set)

Logic and swap instructions

- Bitwise AND, OR and XOR
- Bit test (bitwise AND without storing the result)
- The A type 0111 is the swap (the B type 0111 is the shift)
- Operands: two registers or a register and an 8-bit constant
- Some opcode bits are used as datapath control signals

Machine code	Assembly code	Operation	Z	C	N	V
0100xxxxiiiiiiii	AND rX, #imm	rX ← rX & imm	+	-	+	-
0101xxxxiiiiiiii	OR rX, #imm	rX ← rX | imm	+	-	+	-
0110xxxxiiiiiiii	XOR rX, #imm	rX ← rX ^ imm	+	-	+	-
0111xxxx00000000	SWP rX	rX ← {rX[3:0], rX[7:4]}	+	-	+	-
1000xxxxiiiiiiii	TST rX, #imm	rX & imm	+	-	+	-

Operation select (00: AND, 01: OR, 10: XOR, 11: swap)

1 if the result of the logic operation is not stored

MiniRISC processor - Structure

(Control unit of the MiniRISC CPU – Instruction set)

Logic and swap instructions

- Bitwise AND, OR and XOR
- Bit test (bitwise AND without storing the result)
- The A type 0111 is the swap (the B type 0111 is the shift)
- Operands: two registers or a register and an 8-bit constant
- Some opcode bits are used as datapath control signals

Machine code	Assembly code	Operation	Z	C	N	V
`1111xxxx0100yyyy`	`AND rX, rY`	rX ← rX & rY	+	-	+	-
`1111xxxx0101yyyy`	`OR  rX, rY`	rX ← rX | rY	+	-	+	-
`1111xxxx0110yyyy`	`XOR rX, rY`	rX ← rX ^ rY	+	-	+	-
`1111xxxx1000yyyy`	`TST rX, rY`	rX & rY	+	-	+	-

Operation select (00: AND, 01: OR, 10: XOR)

1 if the result of the logic operation is not stored

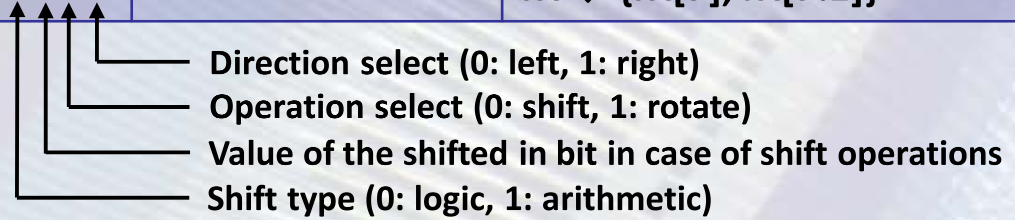
MiniRISC processor - Structure

(Control unit of the MiniRISC CPU – Instruction set)

Shift and rotate instructions

- Logic and arithmetic shift, normal rotate
- Rotate through the carry (C) flag
- Operands: one register (rX) → the rY register address can be used as a control signal, therefore only one opcode is required

Machine code	Assembly code	Operation	Z	C	N	V
`1111xxxx01110000`	`SL0 rX`	rX ← {rX[6:0], 0}	+	+	+	-
`1111xxxx01110100`	`SL1 rX`	rX ← {rX[6:0], 1}	+	+	+	-
`1111xxxx01110001`	`SR0 rX`	rX ← {0, rX[7:1]}	+	+	+	-
`1111xxxx01110101`	`SR1 rX`	rX ← {1, rX[7:1]}	+	+	+	-
`1111xxxx01111001`	`ASR rX`	rX ← {rX[7], rX[7:1]}	+	+	+	-



Direction select (0: left, 1: right)

Operation select (0: shift, 1: rotate)

Value of the shifted in bit in case of shift operations

Shift type (0: logic, 1: arithmetic)

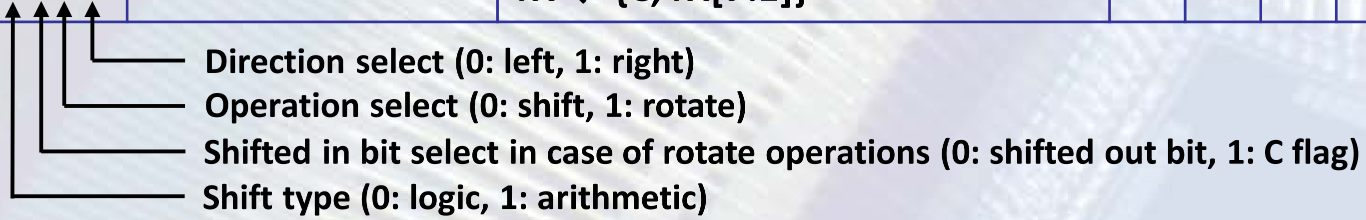
MiniRISC processor - Structure

(Control unit of the MiniRISC CPU – Instruction set)

Shift and rotate instructions

- Logic and arithmetic shift, normal rotate
- Rotate through the carry (C) flag
- Operands: one register (rX) → the rY register address can be used as a control signal, therefore only one opcode is required

Machine code	Assembly code	Operation	Z	C	N	V
`1111xxxx01110010`	`ROL rX`	rX ← {rX[6:0], rX[7]}	+	+	+	-
`1111xxxx01110011`	`ROR rX`	rX ← {rX[0], rX[7:1]}	+	+	+	-
`1111xxxx01110110`	`RLC rX`	rX ← {rX[6:0], C}	+	+	+	-
`1111xxxx01110111`	`RRC rX`	rX ← {C, rX[7:1]}	+	+	+	-



Direction select (0: left, 1: right)

Operation select (0: shift, 1: rotate)

Shifted in bit select in case of rotate operations (0: shifted out bit, 1: C flag)

Shift type (0: logic, 1: arithmetic)

MiniRISC processor - Structure

(Control unit of the MiniRISC CPU – Instruction set)

Program control instructions

- Unconditional jump using absolute or indirect addressing (JMP)
- Conditional jump using absolute or indirect addressing (Jxx)
  - Can be used for testing the value of the ALU status flags
- Operands: a register (rY) or an 8-bit constant
  - The rX register address is not used → selects the operation
  - One opcode is enough for the program control instructions

Machine code	Assembly code	Operation	Z	C	N	V
`10110000aaaaaaaa`	`JMP addr`	PC ← addr	-	-	-	-
`111100001011yyyy`	`JMP (rY)`	PC ← rY	-	-	-	-
`10110001aaaaaaaa`	`JZ addr`	PC ← addr, if Z=1	-	-	-	-
`111100011011yyyy`	`JZ (rY)`	PC ← rY, if Z=1	-	-	-	-
`10110010aaaaaaaa`	`JNZ addr`	PC ← addr, if Z=0	-	-	-	-
`111100101011yyyy`	`JNZ (rY)`	PC ← rY, if Z=0	-	-	-	-

MiniRISC processor - Structure

(Control unit of the MiniRISC CPU – Instruction set)

Program control instructions

- Unconditional jump using absolute or indirect addressing (JMP)
- Conditional jump using absolute or indirect addressing (Jxx)
  - Can be used for testing the value of the ALU status flags
- Operands: a register (rY) or an 8-bit constant
  - The rX register address is not used → selects the operation
  - One opcode is enough for the program control instructions

Machine code	Assembly code	Operation	Z	C	N	V
`10110011aaaaaaaa`	`JC   addr`	PC ← addr, if C=1	-	-	-	-
`111100111011yyyy`	`JC   (rY)`	PC ← rY, if C=1	-	-	-	-
`10110100aaaaaaaa`	`JNC  addr`	PC ← addr, if C=0	-	-	-	-
`111101001011yyyy`	`JNC  (rY)`	PC ← rY, if C=0	-	-	-	-
`10110101aaaaaaaa`	`JN   addr`	PC ← addr, if N=1	-	-	-	-
`111101011011yyyy`	`JN   (rY)`	PC ← rY, if N=1	-	-	-	-

MiniRISC processor - Structure

(Control unit of the MiniRISC CPU – Instruction set)

Program control instructions

- Unconditional jump using absolute or indirect addressing (JMP)
- Conditional jump using absolute or indirect addressing (Jxx)
  - Can be used for testing the value of the ALU status flags
- Operands: a register (rY) or an 8-bit constant
  - The rX register address is not used → selects the operation
  - One opcode is enough for the program control instructions

Machine code	Assembly code	Operation	Z	C	N	V
10110110aaaaaaaa	JNN addr	PC ← addr, if N=0	-	-	-	-
111101101011yyyy	JNN (rY)	PC ← rY, if N=0	-	-	-	-
10110111aaaaaaaa	JV addr	PC ← addr, if V=1	-	-	-	-
111101111011yyyy	JV (rY)	PC ← rY, if V=1	-	-	-	-
10111000aaaaaaaa	JNV addr	PC ← addr, if V=0	-	-	-	-
111110001011yyyy	JNV (rY)	PC ← rY, if V=0	-	-	-	-

MiniRISC processor - Structure

(Control unit of the MiniRISC CPU – Instruction set)

Program control instructions

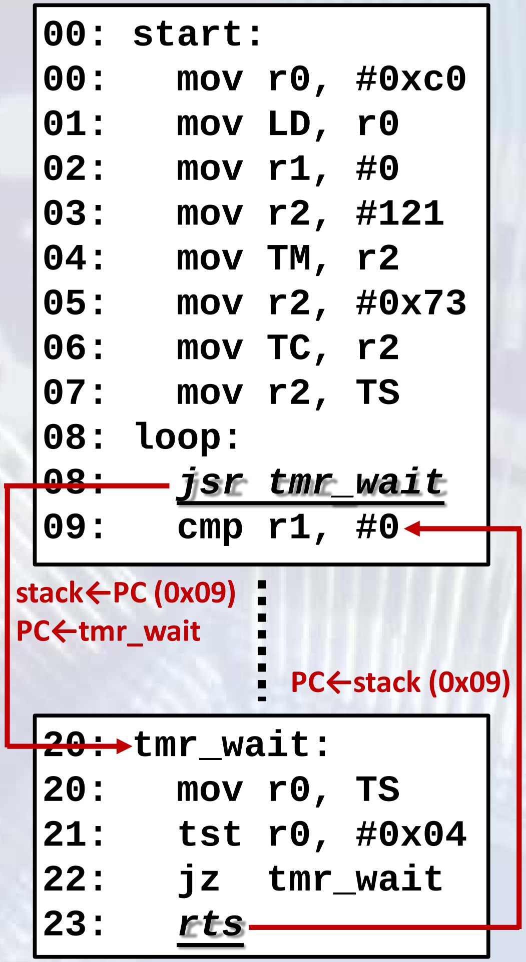
- Subroutine call using absolute or indirect addressing (JSR)
- Return from subroutine (RTS) and from interrupt (RTI)
- Enabling (STI) and disabling (CLI) interrupts
- Operands: a register (rY) or an 8-bit constant
  - The rX register address is not used → selects the operation
  - One opcode is enough for the program control instructions

Machine code	Assembly code	Operation	Z	C	N	V
`10111001aaaaaaaa`	`JSR addr`	stack ← PC ← addr	-	-	-	-
`1111100110 11yyyy`	`JSR (rY)`	stack ← PC ← rY	-	-	-	-
`1011101000000000`	`RTS`	PC ← stack	-	-	-	-
`1011101100000000`	`RTI`	{PC, Z, C, N, V, IE, IF} ← stack	+	+	+	+
`1011110000000000`	`CLI`	IE ← 0	-	-	-	-
`1011110100000000`	`STI`	IE ← 1	-	-	-	-

MiniRISC processor - Structure

(Control unit of the MiniRISC CPU – Subroutine call)

- **Subroutine**
  - Part of the code for executing a given task
  - Relatively independent from the other parts of the code
  - Can be used multiple times → only one instance is necessary
  - Special subroutine call and return instructions
- **Subroutine call: *JSR instruction***
  - Saves the address of the next instruction (return address) to the ***stack***
  - Loads the address of the first instruction of the subroutine to the program counter
- **Return from subroutine: *RTS instruction***
  - Loads the return address from the stack to the program counter



```
00: start:
00:    mov r0, #0xc0
01:    mov LD, r0
02:    mov r1, #0
03:    mov r2, #121
04:    mov TM, r2
05:    mov r2, #0x73
06:    mov TC, r2
07:    mov r2, TS
08: loop:
08:    jsr tmr_wait
09:    cmp r1, #0
```

stack←PC (0x09)
PC←tmr_wait

PC←stack (0x09)

```
20: tmr_wait:
20:    mov r0, TS
21:    tst r0, #0x04
22:    jz  tmr_wait
23:    rts
```

MiniRISC processor - Structure

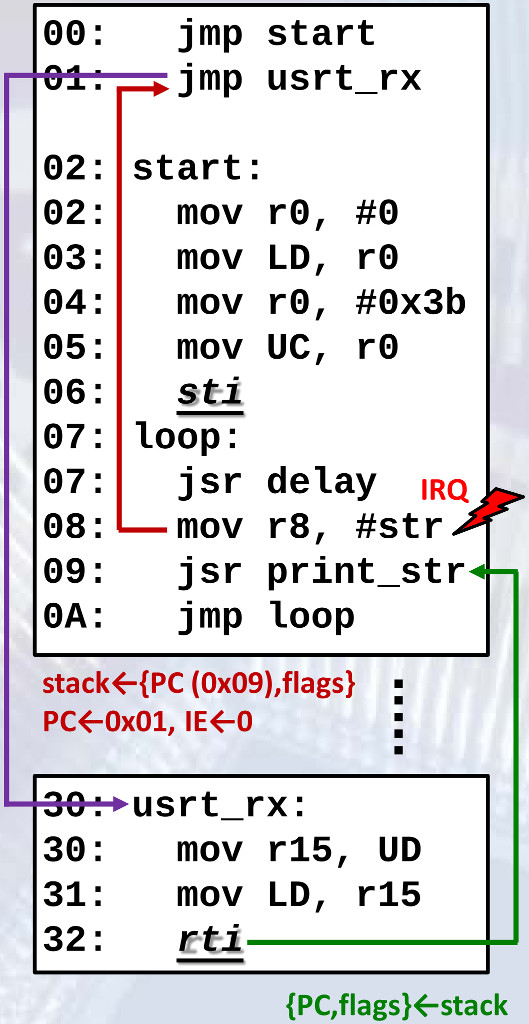
(Control unit of the MiniRISC CPU – Interrupt)

- **The instruction execution happens in the order determined by the programmer**
  - Event handling by polling → slow
  - Many times, faster event handling is required → interrupt
- **Interrupt**
  - External request for service
  - The CPU can accept it after executing the current instruction
- **Interrupt related bits in the control unit of the MiniRISC CPU**
  - ***IE (Interrupt Enable) bit:*** enables the interrupts
    - IE=0: the interrupts are disabled (CLI instruction)
    - IE=1: the interrupts are enabled (STI instruction)
  - ***IF (Interrupt Flag) bit:*** indicates the processor state
    - IF=0: normal program execution
    - IF=1: interrupt service is in progress
    - The value of rhe IF bit is available only through the debug interface

MiniRISC processor - Structure

(Control unit of the MiniRISC CPU – Interrupt)

- **Interrupt system of the MiniRISC CPU**
  - Active-high interrupt request input (IRQ)
  - Simple interrupt system
    - Only the signaling comes from the peripheral, identification of the requester is done in software
    - The address of the interrupt handler (ISR) is fixed 0x01
- **Interrupt servicing in case of the MiniRISC CPU**
  - Similar to the subroutine call
  - If IE=1 and IRQ=1, after executing the current instruction
    - The return address and the flags (ALU status bits, IE, IF) are saved to the ***stack***
    - The interrupt vector (0x01) is loaded into the program counter and the IE bit is cleared
- **Returning from interrupt: *RTI instruction***
  - The PC is loaded with the return address and the flags are restored from the ***stack***



```
00:   jmp start
01:   jmp usrt_rx

02: start:
02:   mov r0, #0
03:   mov LD, r0
04:   mov r0, #0x3b
05:   mov UC, r0
06:   sti
07: loop:
07:   jsr delay
08:   mov r8, #str
09:   jsr print_str
0A:   jmp loop
```

stack←{PC (0x09),flags}
PC←0x01, IE←0

```
30: usrt_rx:
30:   mov r15, UD
31:   mov LD, r15
32:   rti
```

{PC,flags}←stack

MiniRISC processor - Structure

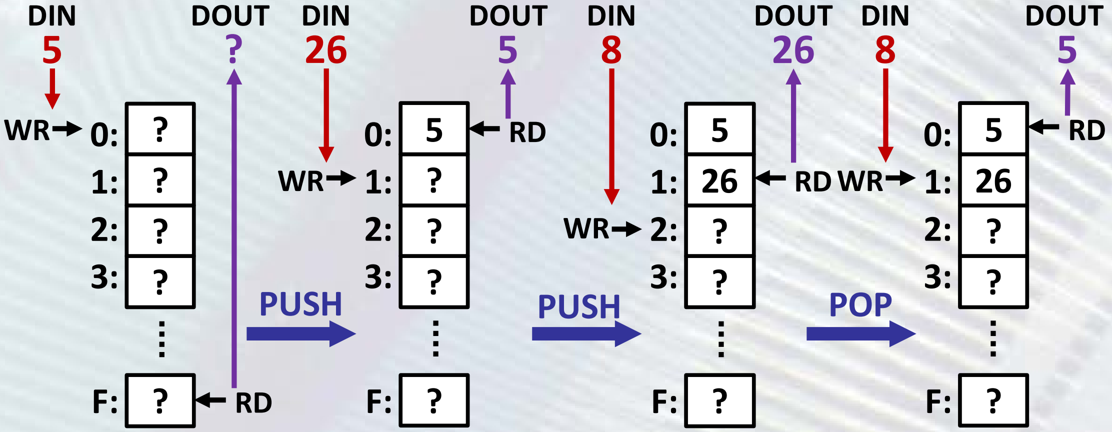
(Control unit of the MiniRISC CPU – Stack)

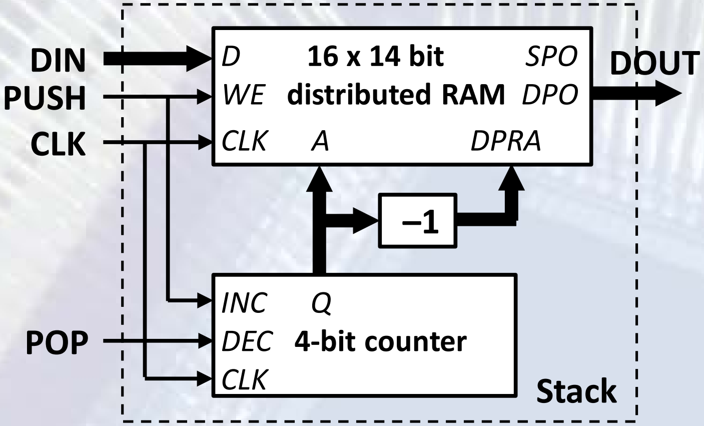
- **The *stack* is a LIFO (Last-In-First-Out) data storage element**
  - The last written data can be read first
  - Two operations are supported
    - ***Push:*** writing data to the top of the stack (full stack→overflow)
    - ***Pop:*** reading data from the top of the stack (empty stack→underflow)
- **The type of the stack can be**
  - Hardware stack inside the processor
  - External stack implemented in the data memory
    - SP (Stack Pointer) register stores the address of the top of the stack
- **The MiniRISC processor contains a 16-word hardware stack**
  - 16-level subroutine call and interrupt service is possible
    - Saved to the stack: program counter (PC), flags (Z, C, N, V, IE and IF)
  - RTS instruction: restores the saved PC value
  - RTI instruction: restores the saved PC value and the flags

MiniRISC processor - Structure

(Control unit of the MiniRISC CPU – Stack)

- **A possibile FPGA implementation of the hardware stack**
  - PC: 8-bit, flags: 6-bit → 16 x 14 bit distributed RAM
    - The push operation enables the write
  - 4-bit bi-directional address counter
    - The push op. increments, the pop op. decrements its value
  - Read address = write address – 1
- **The overflow and underflow conditions are not handled in hardware, this is the task of the programmer!**





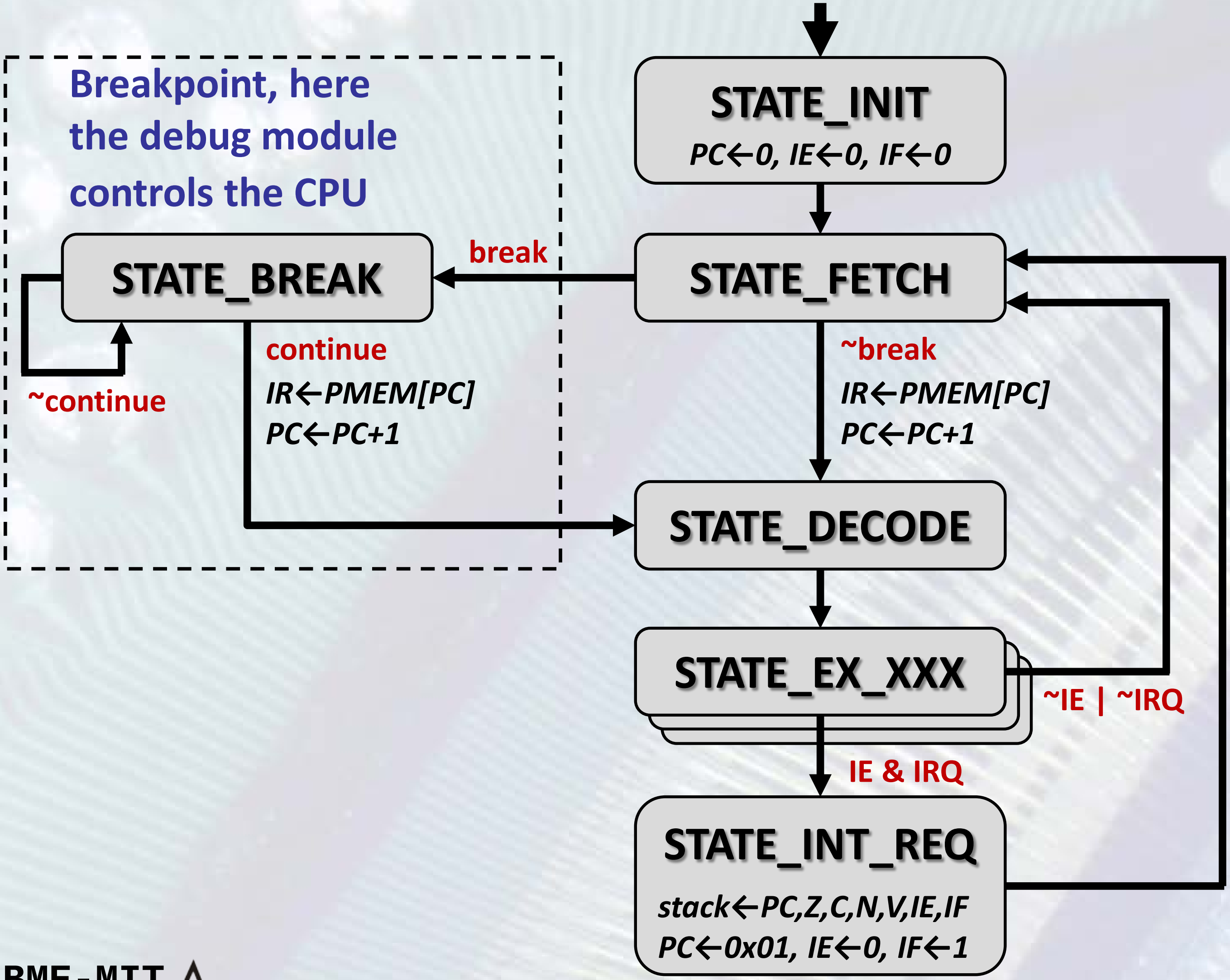
MiniRISC processor - Structure

(Control unit of the MiniRISC CPU – IR and PC)

- **Instruction Register (IR)**
  - 16-bit loadable register
  - In the fetch phase, the instruction register is loaded with the instruction read from the program memory
- **Program Counter (PC)**
  - 8-bit loadable counter with enable
  - Generates the address of the instruction to be fetched
  - Load
    - Processor initialization: loaded with the reset vector (0x00)
    - Interrupt service: loaded with the interrupt vector (0x01)
    - Jump and subroutine call: loaded with the jump address
    - Return (RTS, RTI): loaded with the return address from the stack
  - Enable
    - In the fetch phase, its value is incremented in order to address the next instruction

MiniRISC processor - Structure

(Control unit of the MiniRISC CPU – Controller FSM)



Breakpoint, here the debug module controls the CPU

STATE_BREAK

break

continue

~continue

IR←PMEM[PC]

PC←PC+1

STATE_INIT

PC←0, IE←0, IF←0

STATE_FETCH

~break

IR←PMEM[PC]

PC←PC+1

STATE_DECODE

STATE_EX_XXX

~IE | ~IRQ

IE & IRQ

STATE_INT_REQ

stack←PC,Z,C,N,V,IE,IF

PC←0x01, IE←0, IF←1

Initializing the processor

Instruction fetch

Instruction decode

Instruction execute

Interrupt service

MiniRISC processor - Structure

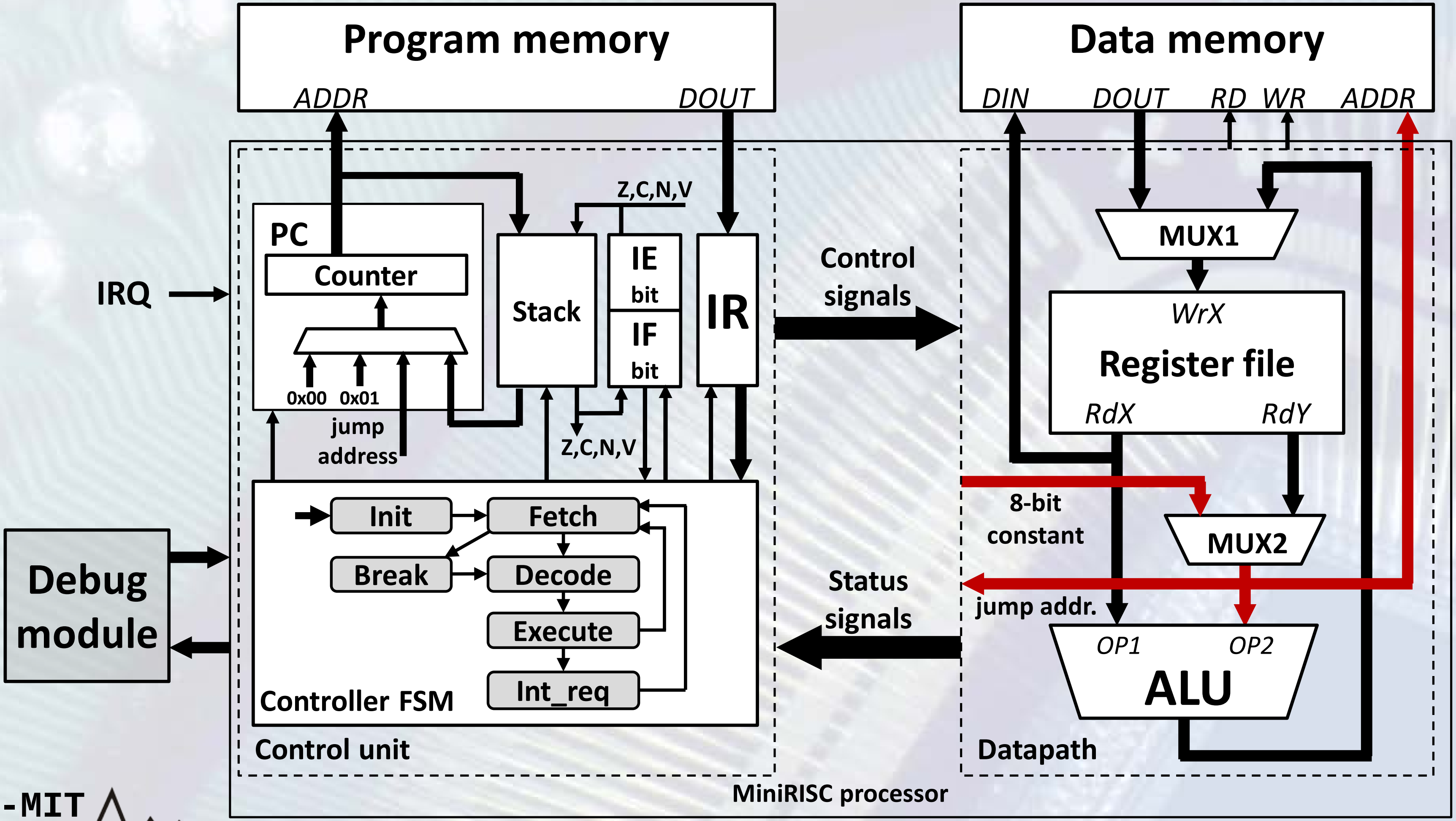
(Control unit of the MiniRISC CPU – Controller FSM)

- **Breakpoint state (*STATE_BREAK*)**
  - Serves software development supporting purposes
  - The debug module takes over the control (details later)
  - ***Break*** signal in the fetch phase → execution stops, BREAK state
  - ***Continue*** signal in the BREAK state → execution resumes
- **Instruction execute**
  - One state is enough for a group of instructions, because some bits of the instructions are used directly as control signals
  - ***STATE_EX_LD:*** executing data memory read (load)
  - ***STATE_EX_ST:*** executing data memory write (store)
  - ***STATE_EX_MOV:*** load constant, move data from register to register
  - ***STATE_EX_ARITH:*** executing arithmetic operations
  - ***STATE_EX_LOGIC:*** executing logic and swap operations
  - ***STATE_EX_SHIFT:*** executing shift and rotate operations
  - ***STATE_EX_CTRL:*** executing program control instructions
  - ***STATE_EX_NOP:*** no operation (for the unused opcodes)

MiniRISC processor - Structure

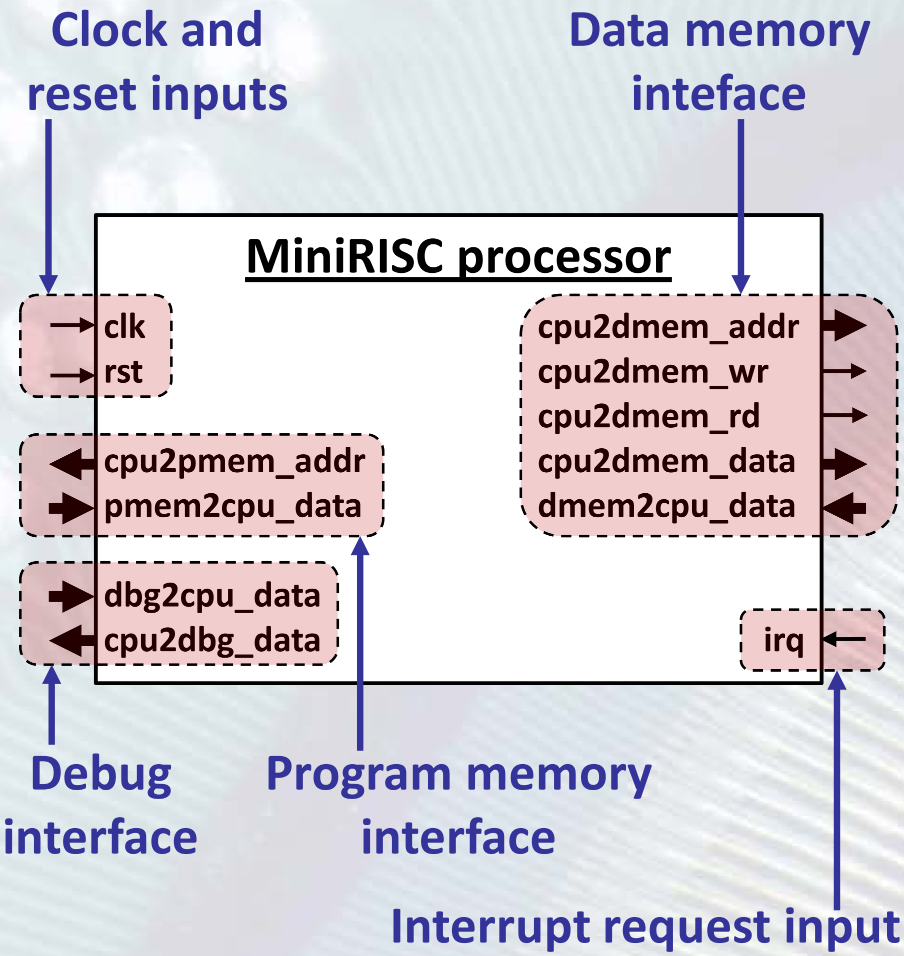
(Block diagram of the MiniRISC processor)

The detailed structure can be found in the Verilog source code



Contents

MiniRISC processor – Interfaces



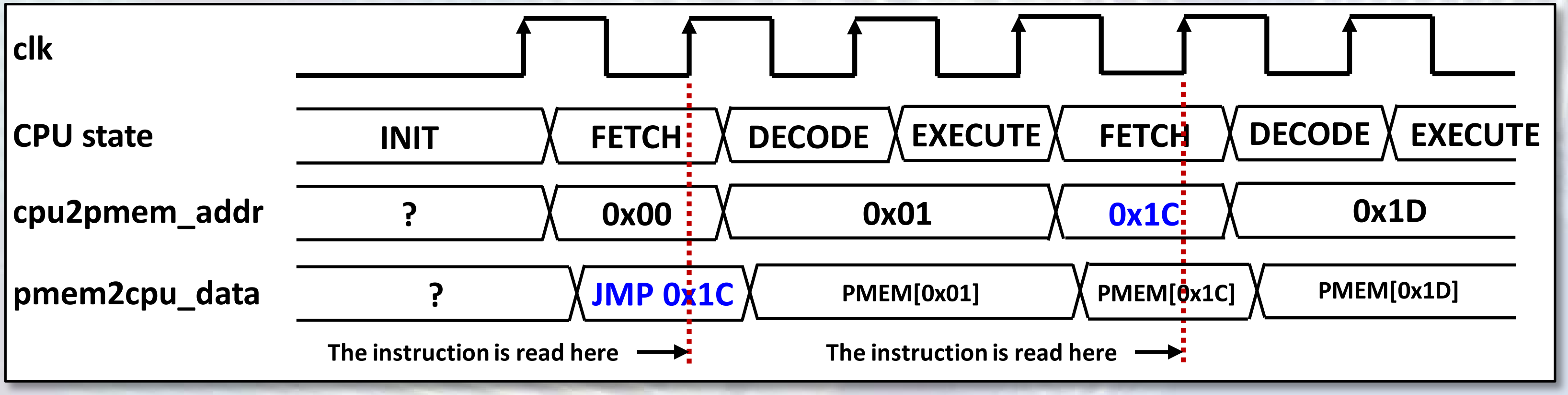
Verilog module header:

```
module minirisc_cpu(
   //Clock and reset.
   input  wire        clk,
   input  wire        rst,

   //Program memory interface.
   output wire [7:0]  cpu2pmem_addr,
   input  wire [15:0] pmem2cpu_data,

   //Data memory interface.
   output wire [7:0]  cpu2dmem_addr,
   output wire        cpu2dmem_wr,
   output wire        cpu2dmem_rd,
   output wire [7:0]  cpu2dmem_data,
   input  wire [7:0]  dmem2cpu_data,

   //Interrupt request input.
   input  wire        irq,

   //Debug interface.
   input  wire [22:0] dbg2cpu_data,
   output wire [44:0] cpu2dbg_data
);
```

MiniRISC processor – Interfaces

(Clock, reset, program memory interface)

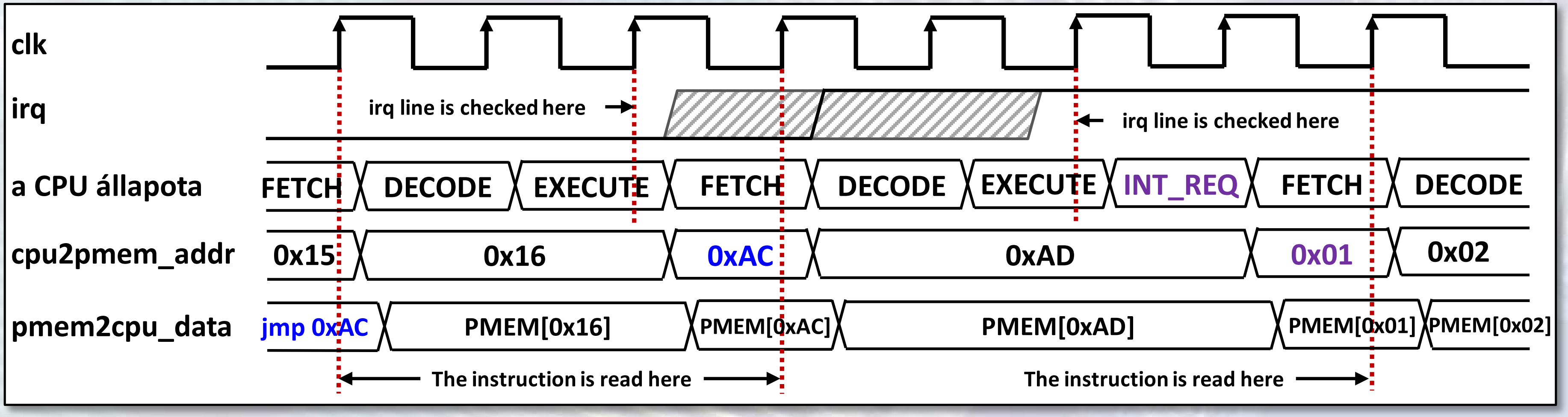
- **Clock and reset**
  - ***clk:*** system clock (16 MHz), every storage element (flip-flop, memory) operates at the rising clock edge
  - ***rst:*** active-high reset signal, sets the CPU to the initial state
- **Program memory interface**
  - ***cpu2pmem_addr:*** 8-bit address bus for the program memory
  - ***pmem2cpu_data:*** 16-bit data output of the program memory
  - The instructions are read in the fetch phase
  - In case of jump, the value of the PC can change in the execute phase
    - The new valid address appears just in the right time on the address bus



MiniRISC processor – Interfaces

(Program memory interface)

- **Program memory interface**
  - In case of jump or subroutine call, the value of the program counter can be changed in the execution phase
  - When an interrupt is serviced, the interrupt vector address (0x01) is loaded into the program counter in the INT_REQ state
  - The new programm memory address is available in time for the next instruction fetch phase



MiniRISC processor – Interfaces

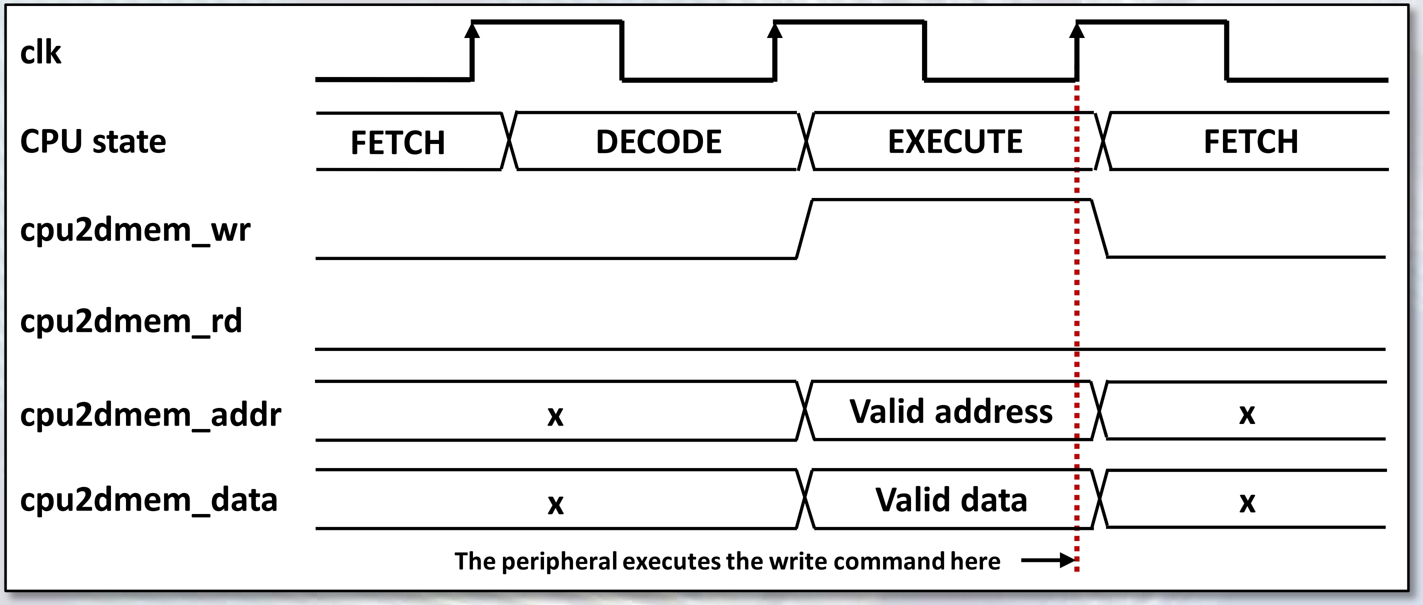
(Data memory interface)

- **Data memory interface**
  - Simple synchronous bus, commands are valid at the rising clock edge
  - ***cpu2dmem_addr:*** 8-bit address bus
  - ***cpu2dmem_wr:*** active-high write enable signal
  - ***cpu2dmem_rd:*** active-high read enable signal
  - ***cpu2dmem_data:*** 8-bit write data bus (CPU → peripheral)
  - ***dmem2cpu_data:*** 8-bit read data bus (peripheral → CPU)
- **There are no tri-state drivers inside the FPGA, therefore two separate data bus is required (wrong control of the tri-state drivers would cause short circuit which would damadge the FPGA)**
- **The MiniRISC processor doesn’t have separate I/O interface for the peripherals, therefore the peripherals can be connected to the data memory interface (the peripherals are embedded in the data memory)**
- **In case of multiple connected peripherals, if the inactive peripherals drive the read data bus with 0, the read data buses can be ORed together and no multiplexer is required**
  - Distributed bus multiplexer function

MiniRISC processor – Interfaces

(Data memory interface – Write cycle)

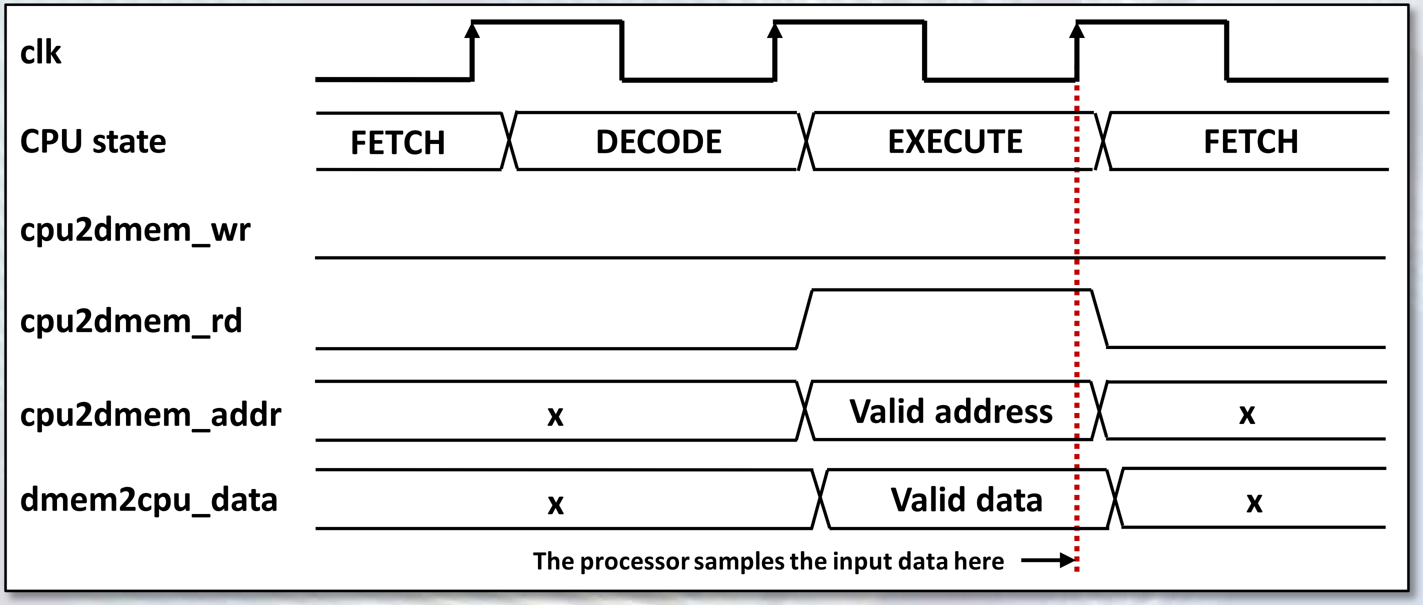
- **Write cycle of the data memory interface**
  - In the execute phase, the write cycle is indicated by the ***cpu2dmem_wr*** signal which is active for 1 clock cycle
  - During the write cycle, the ***cpu2dmem_addr*** address is stable
  - During the write cycle, the ***cpu2dmem_data*** data is stable, which is sampled by the selected peripheral at the rising edge of the clock



MiniRISC processor – Interfaces

(Data memory interface – Read cycle)

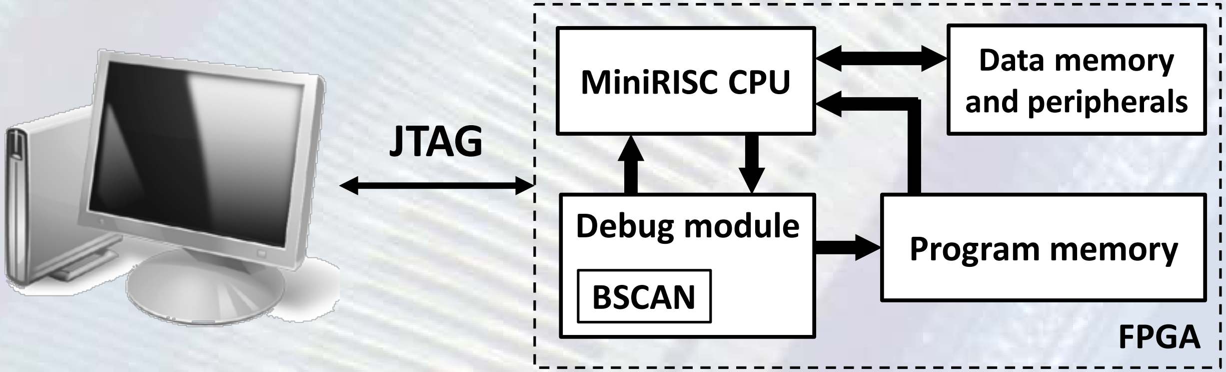
- **Read cycle of the data memory interface**
  - In the execute phase, the read cycle is indicated by the ***cpu2dmem_rd*** signal which is active for 1 clock cycle
  - During the read cycle, the ***cpu2dmem_addr*** address is stable
  - During the read cycle, the selected peripheral drives the ***dmem2cpu_data*** read data bus with the valid data, the other peripherals drive their data outputs with inactive 0 value



MiniRISC processor – Interfaces

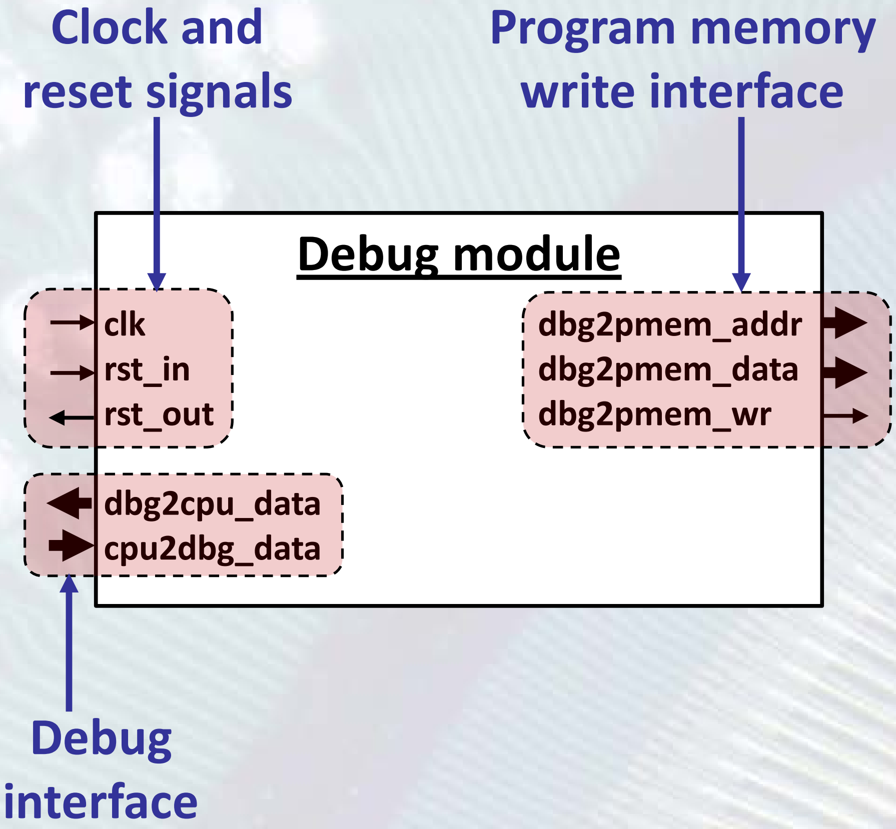
(Debug module, debug interface)

- **Debug module: supports the software development**
  - Gives reset signal to the processor system
  - Program download (program memory write)
  - Register file, PC and ALU status flags read and write
  - Data memory read and write
  - Place breakpoint to any program memory address
    - Stepping into a breakpoint suspends the execution of the program
  - Suspend and resume the execution of the program
- **The communication between the debug module and the MiniRISC development environment uses the JTAG interface**



MiniRISC processor – Interfaces

(Debug module, debug interface)



Verilog module header:

```
module debug_module(
   //Clock and reser.
   input  wire        clk,
   input  wire        rst_in,
   output wire        rst_out,

   //Program memory write interface.
   output wire [7:0]  dbg2pmem_addr,
   output wire [15:0] dbg2pmem_data,
   output wire        dbg2pmem_wr,

   //Debug interface.
   output wire [22:0] dbg2cpu_data,
   input  wire [44:0] cpu2dbg_data
);
```

MiniRISC processor – Interfaces

(Debug module, debug interface)

- **Interface between the MiniRISC CPU and the debug module**
  - ***dbg2cpu_data:*** signals from the debug module to the CPU
  - ***cpu2dbg_data:*** signals from the CPU to the debug module
  - The ***dbg2cpu_data*** input should be driven with 0 if no debug module is in the system
- **Other signals of the debug module**
  - Clock and reset
    - ***clk:*** system clock (16MHz)
    - ***rst_in:*** external reset signal (for example: reset button)
    - ***rst_out:*** reset signal for the processor system
  - Program memory write interface
    - ***dbg2pmem_addr:*** 8-bit address bus
    - ***dbg2pmem_data:*** 16-bit write data bus
    - ***dbg2pmem_wr:*** active-high write enable signal

MiniRISC processor – I/O extension

Steps of the I/O extension task

- **Collecting the requirements according to the type of the peripheral**
  - Number of registers, their usage mode (writable, readable)
    - Command, status, mode select, etc. registers
  - Maybe FIFO or small memory block
- **Base address assignment, designing the usage of the address range**
- **Address decoding**
  - ***psel = ((cpu2dmem_addr >> N) == (BASEADDR >> N))***
  - Size of the address range is 2^N bytes
- **Write enable signals**
  - ***xxx_wr = psel & cpu2dmem_wr & (cpu2dmem_addr[N-1:0] == ADDR)***
- **Read enable signals**
  - ***xxx_rd = psel & cpu2dmem_rd & (cpu2dmem_addr[N-1:0] == ADDR)***
  - Controls the output MUX: only one multiplexer output is valid at a time, the other peripherals drive their data output with inactive 0
  - Also requires when the read causes state change (for example: FIFO)

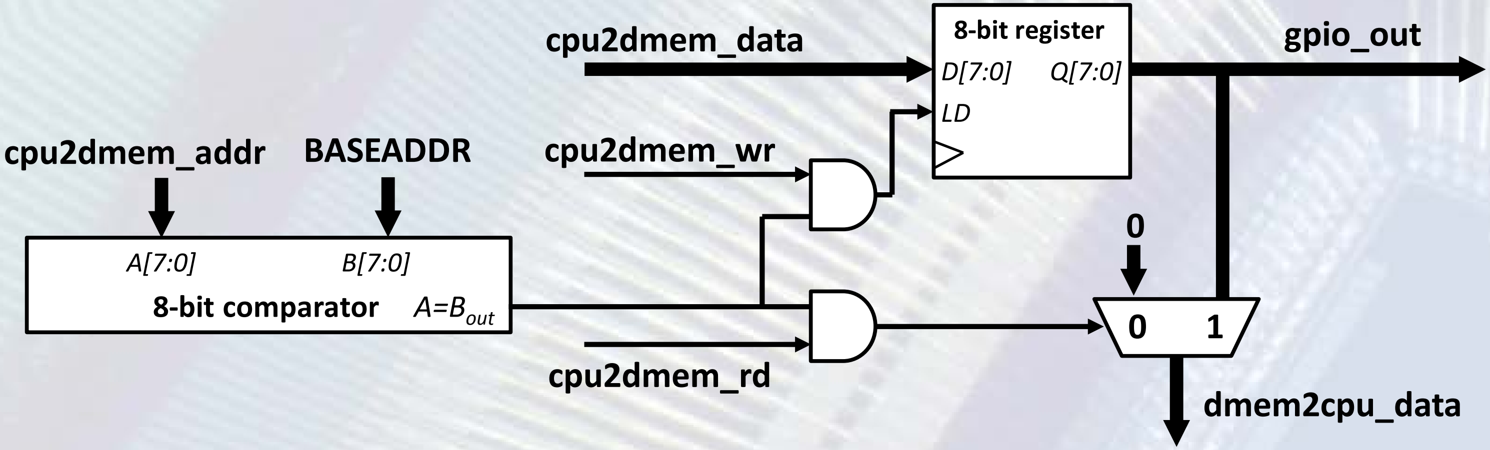
Checking the lower address bits is required, if N > 0

MiniRISC processor – I/O extension

(Examples)

Example 1: 8-bit output peripheral with readback

- **Can be used to drive the LEDs on the FPGA board**
- **Very simple**
  - A register to store the output data
    - BASEADDR + 0x00, 8-bit, writable and readable
  - Address decoding logic
    - 1 register → 1-byte address range is required



MiniRISC processor – I/O extension

(Examples)

Example 1: 8-bit output peripheral with readback

```
module basic_owr #(
   //Base address of the peripheral.
   parameter BASEADDR = 8'hff
) (
   //Clock and reset.
   input  wire       clk,
   input  wire       rst,

   //Data memory interface.
   input  wire [7:0] cpu2dmem_addr,
   input  wire       cpu2dmem_wr,
   input  wire       cpu2dmem_rd,
   input  wire [7:0] cpu2dmem_data,
   output reg  [7:0] dmem2cpu_data,

   //Output data.
   output reg  [7:0] gpio_out
);

//Select signal of the peripheral.
wire psel = (cpu2dmem_addr == BASEADDR);
```

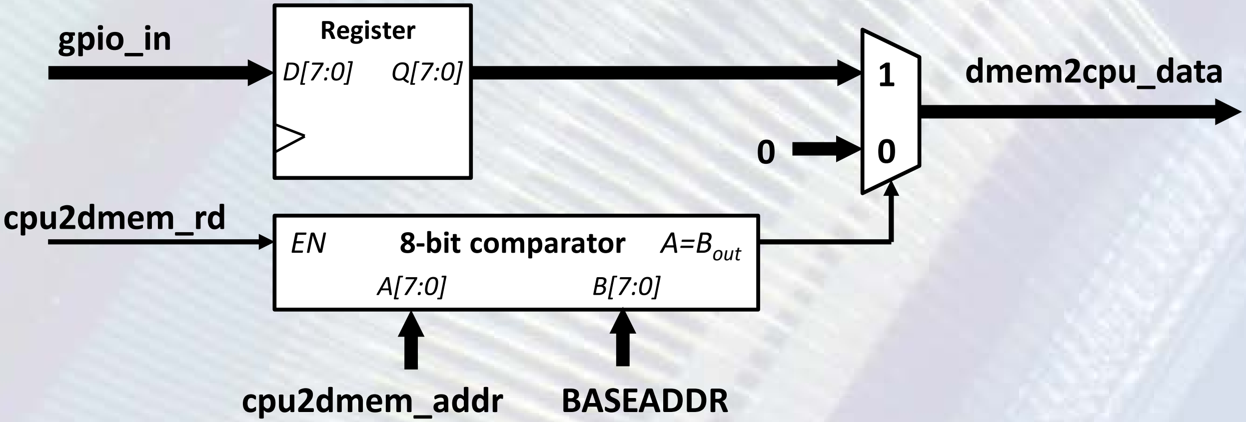
```
//Data reg. write enable signal.
wire dreg_wr = psel & cpu2dmem_wr;

//Data reg. read enable signal.
wire dreg_rd = psel & cpu2dmem_rd;

//Output data register.
always @(posedge clk)
   if (rst)
      gpio_out <= 8’d0;
   else
      if (dreg_wr)
         gpio_dout <= cpu2dmem_data;

//Driving the read data bus.
always @(*)
   if (dreg_rd)
      dmem2cpu_data <= gpio_dout;
   else
      dmem2cpu_data <= 8’d0;

endmodule
```

MiniRISC processor – I/O extension

(Examples)

Example 2: **8-bit input peripheral**

- **Can be used to connect switches or buttons to the system**
- **Very simple**
  - A register to sample the input data
    - BASEADDR + 0x00, 8-bit, read-only
  - Address decoding logic
    - 1 register → 1-byte address range is required



MiniRISC processor – I/O extension

(Examples)

Example 2: 8-bit input peripheral

```
module basic_in #(
   //Base address of the peripheral.
   parameter BASEADDR = 8'hff
) (
   //Clock and reset.
   input  wire       clk,
   input  wire       rst,

   //Data memory interface.
   input  wire [7:0] cpu2dmem_addr,
   input  wire       cpu2dmem_rd,
   output reg  [7:0] dmem2cpu_data,

   //Input data.
   input  wire [7:0] gpio_in
);

//Select signal of the peripheral.
wire psel = (cpu2dmem_addr == BASEADDR);
```

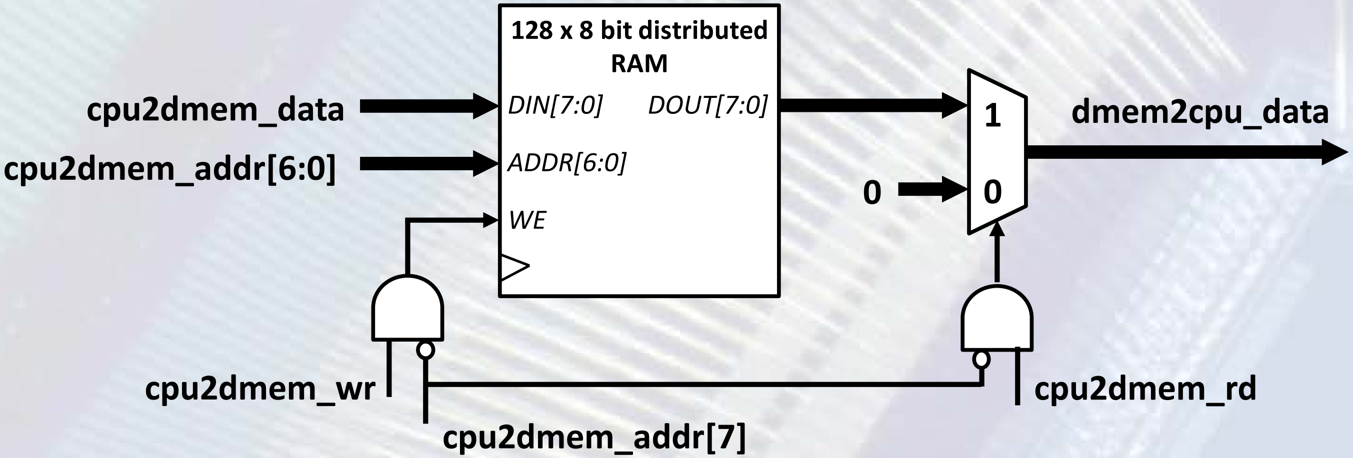
```
//Data reg. read enable signal.
wire in_reg_rd = psel & cpu2dmem_rd;

//Input data register.
reg [7:0] in_reg;

always @(posedge clk)
   if (rst)
      in_reg <= 8’d0;
   else
      in_reg <= gpio_in;

//Driving the read data bus.
always @(*)
   if (in_reg_rd)
      dmem2cpu_data <= in_reg;
   else
      dmem2cpu_data <= 8’d0;

endmodule
```

MiniRISC processor – I/O extension

(Examples)

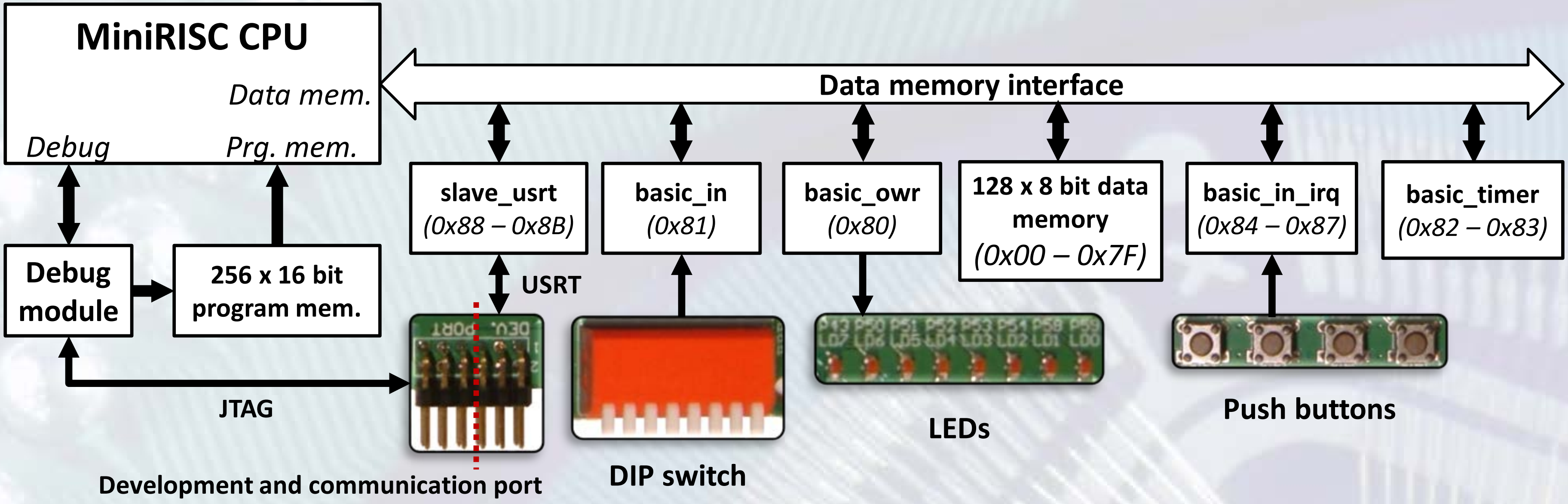
Example 3: 128 x 8 bit data memory

- **Requires 7 address bits**
  - Address range: **00000000 (0x00) – 01111111 (0x7F)**
- **Address decoding logic**
  - The lower 7 address bits connected to the memory
  - The upper address bit (MSb) is used for address decoding
    - If it is 0, the memory is selected



MiniRISC system

(Simplified MiniRISC system – Block diagram)



MiniRISC CPU

Data mem.

Debug

Prg. mem.

Data memory interface

slave_usrt *(0x88 – 0x8B)*

basic_in *(0x81)*

basic_owr *(0x80)*

128 x 8 bit data memory *(0x00 – 0x7F)*

basic_in_irq *(0x84 – 0x87)*

basic_timer *(0x82 – 0x83)*

Debug module

256 x 16 bit program mem.

USRT

JTAG

Development and communication port

DIP switch

LEDs

Push buttons

Address range	Size	Peripheral	Function
0x00 – 0x7F	128 bytes	data memory	128 x 8 bit memory
0x80	1 byte	basic_owr	interfacing the LEDs
0x81	1 byte	basic_in	interfacing the DIP switch
0x82 – 0x83	2 bytes	basic_timer	timing
0x84 – 0x87	4 bytes	basic_in_irq	Interfacing the push-buttons
0x88 – 0x8B	4 bytes	slave_usrt	serial communication

MiniRISC system

(Peripherals – basic_owr and basic_in)

- ***basic_owr:*** **8-bit output peripheral with readback**
  - Simple output, initial value is 0x00
  - Data register: BASEADDR + 0x00, writable and readable
    - The OUT$_i$ bit of the data reg. sets the value of the i-th output bit

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
OUT7	**OUT6**	**OUT5**	**OUT4**	**OUT3**	**OUT2**	**OUT1**	**OUT0**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

  - Used for interfacing the LEDs
- ***basic_in:*** **8-bit input peripheral**
  - Simple input with continuous sampling
  - Data register: BASEADDR + 0x00, read-only
    - The IN$_i$ bit of the data reg. gives the state of the i-th input bit

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
IN7	**IN6**	**IN5**	**IN4**	**IN3**	**IN2**	**IN1**	**IN0**
R	R	R	R	R	R	R	R

  - Used for interfacing the DIP switch and the push buttons

MiniRISC system
(Peripherals – basic_in_irq)

basic_in_irq: **8-bit input peripheral with debouncing and interrupts**

- **Data register**
  - BASEADDR + 0x00, 8-bit, read-only
  - The IN_i data register bit gives the state of the i-th input bit

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
IN7	IN6	IN5	IN4	IN3	IN2	IN1	IN0
R	R	R	R	R	R	R	R

- **Interrupt enable register (IE)**
  - BASEADDR + 0x01, 8-bit, writable and readable
  - The IE_i bit enables the interrupt request for the change of the i-th input bit

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
IE7	IE6	IE5	IE4	IE3	IE2	IE1	IE0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

- **Interrupt flag register (IF)**
  - BASEADDR + 0x02, 8-bit, writable and readable
  - The IF_i bit indicates the change of the i-th input, write 1 to clear the flag

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
IF7	IF6	IF5	IF4	IF3	IF2	IF1	IF0
R/W1C	R/W1C	R/W1C	R/W1C	R/W1C	R/W1C	R/W1C	R/W1C

- **Used for interfacing the 4 push-buttons (the upper 4 register bits are not used)**

MiniRISC system

(Peripherals – basic_timer)

basic_timer: **timer peripheral**

- **Structure: a clock prescaler and an 8-bit down-counter**
  - The clock prescaler enables the timer counter at given intervals
- **Counter initial state register (TR)**
  - BASEADDR + 0x00, 8-bit, write-only
  - The initial state of the counter determines the timer period

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
W	W	W	W	W	W	W	W

- **Counter register (TM)**
  - BASEADDR + 0x00, 8-bit, read-only
  - Current value of the timer counter

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
TM7	TM6	TM5	TM4	TM3	TM2	TM1	TM0
R	R	R	R	R	R	R	R

MiniRISC system

(Peripherals – basic_timer)

basic_timer: **timer peripheral**

- **Command register (TC): BASEADDR + 0x01, 8-bit, write-only**

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
TIE	TPS2	TPS1	TPS0	-	-	TREP	TEN
W	W	W	W	n.a.	n.a.	W	W

- **Status register (TS): BASEADDR + 0x01, 8-bit, read-only**

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
TIT	TPS2	TPS1	TPS0	0	TOUT	TREP	TEN
R	R	R	R	R	R	R	R

Bit	Function
TEN	Timer enable bit (0: timer is disabled, 1: timer is enabled)
TREP	Timer mode select bit (0: single cycle, 1: repeat)
TOUT	Indicates whether the timer period has been elapsed
TPS[2:0]	Clock prescaler value select bits 0 : no prescale 1 – 7 : $2^{2\cdot(TPS+1)}$ prescale (16, 64, 256, 1024, 4096, 16384 or 65536)
TIE / TIT	Timer interrupt enable / flag (TIT flag is cleared when TS is read)

MiniRISC system

(Peripherals – basic_timer)

basic_timer: **timer peripheral**

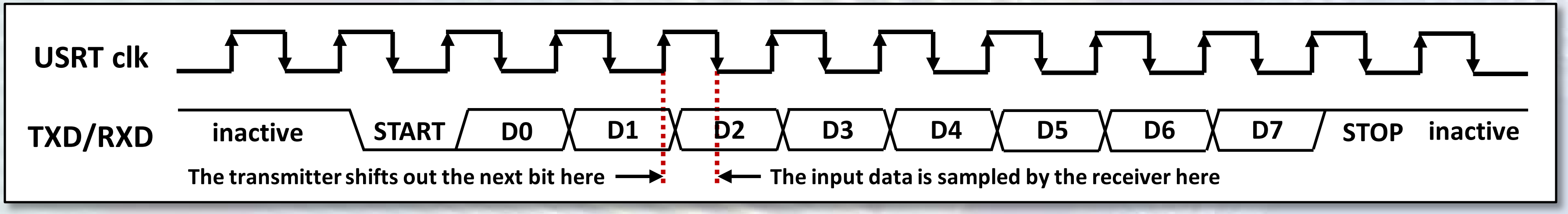
- **Timer period:** $T = (TR + 1) \cdot PS \cdot T_{CLK}$
  - TR is the initial value of the timer counter (0 – 255)
  - PS is the prescale (1, 16, 64, 256, 1024, 4096, 16384 or 65536)
  - T_{CLK} is the system clock period (f_{CLK}=16 MHz → T_{CLK}=62,5 ns)
- **The maximum timer period that can be set is 1,048576 s**
- **After setting the parameters (TR, TPS, TREP), the timer can be started by setting the TEN bit to 1**
- **In case of single cycle mode (TREP=0), the counter stops after reaching the last state (0). In case of repeat mode (TREP=1) the counter is reloaded with TR after reaching the last state.**
- **The TOUT bit indicates that the timer period has been elapsed, this bit can be cleared by reading the status register (TS)**
- **If the interrupt is enabled (TIE=1), the TOUT bit also activates the interrupt request output of the timer**

MiniRISC system

(Peripherals – slave_usrt)

slave_usrt: **peripheral that provides serial communication**

- **USRT (Universal Serial Receiver Transmitter) data transfer**
  - Data framing: 1 START bit (0), 8 data bits, 1 STOP bit (1)
  - USRT clock: determines the communication speed
    - The master device outputs the clock to the slave device
  - Transfer: the new bit is shifted out at the rising edge of the USRT clock
  - Reception: the input is sampled at the falling edge of the USRT clock
    - Only the characters without frame error (STOP bit = 1) are stored



MiniRISC system

(Peripherals – slave_usrt)

slave_usrt: **peripheral that provides serial communication**

- **Control register (UC): BASEADDR + 0x00, 8-bit, writable/readable**

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
0	0	0	0	RXCLR	TXCLR	RXEN	TXEN
R	R	R	R	W	W	R/W	R/W

Bit	Mode	Function
TXEN	R/W	0: USRT transmitter is disabled 1: USRT transmitter is enabled
RXEN	R/W	0: USRT receiver is disabled 1: USRT receiver is enabled
TXCLR	W	Write 1 here to clear the transmit FIFO
RXCLR	W	Write 1 here to clear the receive FIFO

- **Data register (UD): BASEADDR + 0x03, 8-bit, writable/readable**
  - Write: write data to the transmit (TX) FIFO (if TXNF=1)
  - Read: read data from the receive (RX) FIFO (if RXNE=1)

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
D7	D6	D5	D4	D3	D2	D1	D0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

MiniRISC system

(Peripherals – slave_usrt)

slave_usrt: **peripheral that provides serial communication**

- **FIFO status register (US): BASEADDR + 0x01, 8-bit, read-only**

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
0	0	0	0	RXFULL	RXNE	TXNF	TXEMPTY
R	R	R	R	R	R	R	R

- **Interrupt enable reg. (UIE): BASEADDR + 0x02, 8-bit, writable/readable**
  - The FIFO status interrupts can be enabled/disabled here
  - The interrupt request is active while the enabled events are active

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
0	0	0	0	RXFULL	RXNE	TXNF	TXEMPTY
R	R	R	R	R/W	R/W	R/W	R/W

Bit	Meaning	
TXEMPTY	0: the TX FIFO contains data	1: the TX FIFO is empty
TXNF	0: the TX FIFO is full	1: the TX FIFO is not full
RXNE	0: the RX FIFO is empty	1: the RX FIFO contains data
RXFULL	0: the RX FIFO is not full	1: the RX FIFO is full

Contents

1. **Introduction**
2. **Internal structure of the MiniRISC CPU**
   - Datapath
   - Control unit
3. **Application of the MiniRISC CPU**
   - Signal interfaces
   - I/O extension (with examples)
   - MiniRISC system
4. ***Development environment***
   - MiniRISC assembler
   - MiniRISC IDE
   - Software development (with examples)

MiniRISC assembler

- **A development environment (MiniRISC IDE) is available for the MiniRISC system designed for the LOGSYS Spartan-3E FPGA board**
- **The programs written in assembly language can be compiled using the LOGSYS MiniRISC Lite assembler**
  - Requires the .NET Framework 4.0 to run (this is the part of the operating system from Windows 8)
- **The capabilities of the assembler fit the MiniRISC CPU**
  - Simple instruction interpretation
  - Generates absolute code (no linker)
  - No macros
  - No address arithmetic
  - No conditional compilation
  - Stops in case of error

MiniRISC assembler

- **Running from command line: *MiniRISCv2-as filename.s***
  - Reads the ***filename.s*** assembly source file, and in case of no error, the following files are generated:

filename.lst	assembly list file with addresses, labels, identifiers and machine code
code.hex	text file for initializing the program memory from the Verilog source code
data.hex	text file for initializing the data memory from the Verilog source code
filename.svf	SVF file for initializing the program and data memories using the LOGSYS GUI
filename.dbgdat	file containing the debug informations

  - The error messages appear in the console
- **The MiniRISC assembler is integrated with the MiniRISC IDE, therefore it is not common to run the assembler from command line**

MiniRISC assembler

- **Souce file: simple text file with .s extension**
- **Processed line-by-line: one instruction per source code line**
- **Format of the assembly source code lines**
  `LABEL: INSTR OP1{, OP2}    ; Comment`

– ***LABEL***	Identifier representing the address of the following instruction
– ***INSTR***	Mnemonic referring to the operation, for example: ***ADD*** – addition, ***JMP*** – jump, etc.
– ***OP1{, OP2}***	Operands of the instruction, the ***OP2*** is not always present
– ***; Comment***	The ';' character indicates the start of the comment which is skipped by the assembler

- **Recommendations**
  - Write comments for each instruction
  - Use the TAB character for formatting the code

MiniRISC assembler

- **Only few rules exist**
- **The operands of the instructions can be**
  - Register r0 – r15
  - Constant/immediate #0 – #255 (for ALU operations)
  - Memory address 0 – 255 (constant for memory addressing)
  - Indirect address (r0) – (r15)
- **Numeric constants**
  - No prefix decimal 0 – 255
  - 0x prefix hexadecimal 0x00 – 0xFF
  - 0b prefix binary 0b00000000 – 0b11111111
- **Character constants**
  - A character between the ′ ′ marks (example: ***#′A′*** – value: 65)
  - Escape sequences: ***’\’’, ’\”’, ’\\’, ’\a’, ’\b’, ’\f’, ’\n’, ’\r’, ’\t’, ’\v’***
- **String constants**
  - Characters between the ″ ″ marks (example: ***″MiniRISC\r\n″***)
  - Can be used only in the data section

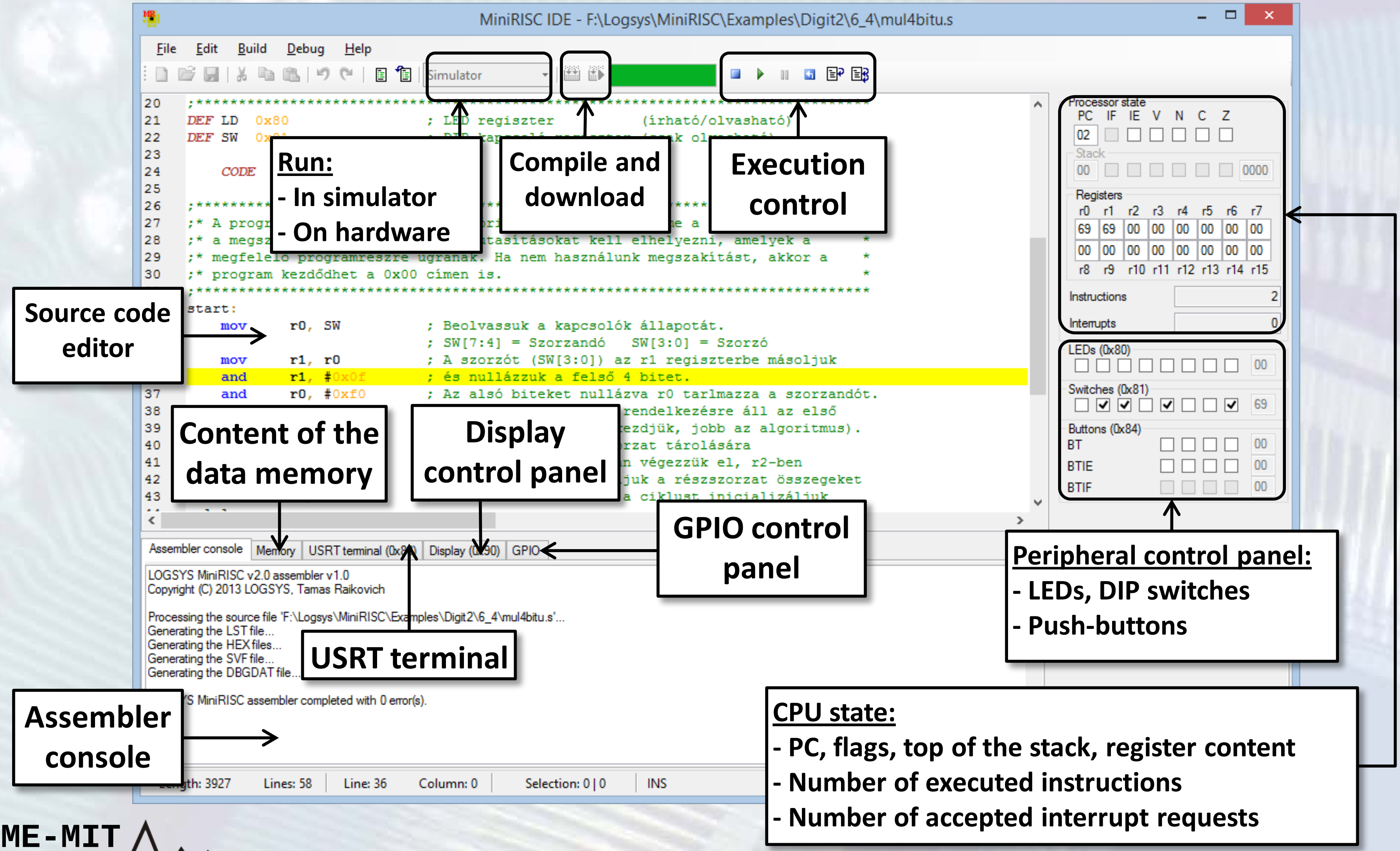
MiniRISC assembler

- **Assembler directives**
  - ***DEF:*** assigns an identifier to a constant
    `DEF SW 0x81 ;Address of the DIP switch`
    - This is not a CPU instruction. It defines a replacement rule for the assembler, which provides better readability of the source code for the user.
  - ***CODE:*** indicates the beginning of the code section
    - The generated code is placed into the program memory
  - ***DATA:*** indicates the beginning of the data section
    - The generated code is placed into the data memory
    - Only labels and DB directives are allowed in the data section
  - ***DB:*** initializes the data memory with constants
    `DB ”MiniRISC.\r\n”, 0  ;0 terminated string`
    - Can be used only in the data section
    - Numeric, character and string constants can follow the DB directive
    - The constants are separated with the comma character

MiniRISC assembler

- **Assembler directives**
  - ***ORG:*** directly defines the start address
    `ORG memory_address`
    - Sets the start address of the following code segment
    - Can be used in both code and data sections
- **The address and the machine code of the instructions, and the interpretation of the identifiers used by the compiler can be checked in the generated LST file**
- **If the program has been compiled without errors, it can be built in to the design as a memory initializer data (.HEX output files) or it can be downloaded to the already configured FPGA device (.SVF output file)**

MiniRISC IDE



```
;****************************************************************************
DEF LD   0x80                 ; LED regiszter          (írható/olvasható)
DEF SW   0x


    CODE


;*********
;* A progr
;* a megsz                          tasításokat kell elhelyezni, amelyek a     *
;* megfelelő programrészre ugranak. Ha nem használunk megszakítást, akkor a     *
;* program kezdődhet a 0x00 címen is.                                          *
;****************************************************************************
start:
    mov       r0, SW              ; Beolvassuk a kapcsolók állapotát.
                                  ; SW[7:4] = Szorzandó    SW[3:0] = Szorzó
    mov       r1, r0              ; A szorzót (SW[3:0]) az r1 regiszterbe másoljuk
    and       r1, #0x0f           ; és nullázzuk a felső 4 bitet.
    and       r0, #0xf0           ; Az alsó biteket nullázva r0 tarlmazza a szorzandót.
                                          rendelkezésre áll az első
                                          ezdjük, jobb az algoritmus).
                                          rzat tárolására
                                          n végezzük el, r2-ben
                                          juk a részszorzat összegeket
                                          a ciklust inicializáljuk
```

```
LOGSYS MiniRISC v2.0 assembler v1.0
Copyright (C) 2013 LOGSYS, Tamas Raikovich

Processing the source file 'F:\Logsys\MiniRISC\Examples\Digit2\6_4\mul4bitu.s'...
Generating the LST file...
Generating the HEX files...
Generating the SVF file...
Generating the DBGDAT file...

YS MiniRISC assembler completed with 0 error(s).
```

Run:
- In simulator
- On hardware

Compile and download

Execution control

Source code editor

Content of the data memory

Display control panel

GPIO control panel

USRT terminal

Assembler console

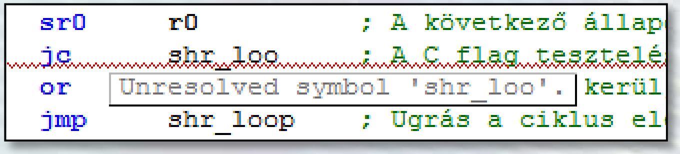
Peripheral control panel:
- LEDs, DIP switches
- Push-buttons

CPU state:
- PC, flags, top of the stack, register content
- Number of executed instructions
- Number of accepted interrupt requests

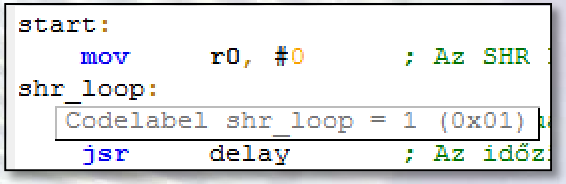
MiniRISC IDE

Capabilities of the source code editor:

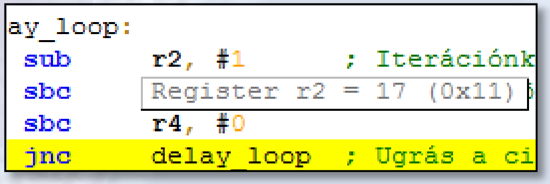
- **Syntax highlighting**
- **Cut, copy, paste (✂ 📄 📋), undo, redo (↶ ↷)**
- **Comment and uncomment of the selected lines (📄 📄)**
- **Underlining the wrong source code lines, and displaying the error message if the cursor points to a wrong line**
- **In case of a compiled program, displaying the value of the identifiers if the cursor points to an identifier**
- **In case of a downloaded program**
  - Displaying the value stored in the registers if the cursor points to a register
  - In case of indirect addressing, displaying the address and the data if the cursor points to a register



```
sr0     r0          ; A következő állap
jc      shr_loo     ; A C flag tesztelé
or   Unresolved symbol 'shr_loo'. kerül
jmp     shr_loop    ; Ugrás a ciklus el
```

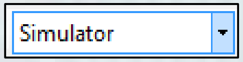


```
start:
    mov     r0, #0          ; Az SHR
shr_loop:
   Codelabel shr_loop = 1 (0x01)
    jsr     delay           ; Az idő
```



```
ay_loop:
 sub      r2, #1         ; Iterációnk
 sbc      Register r2 = 17 (0x11)
 sbc      r4, #0
 jnc      delay_loop  ; Ugrás a ci
```

MiniRISC IDE

- **From the  dropdown menu, it can be selected whether the program should run**

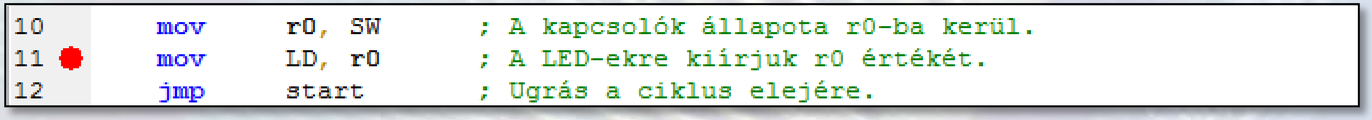
  - In the simulator (this option is always available)
  - On the hardware (***LDCxxx*** development cable)
  - Selection is only possible when the program doesn’t run
- **Simulator**
  - Simulates the processor system implemented in the FPGA with clock cycle precision
  - The program execution is slower in this case
    - ~400000 instruction/s → **~1,2 MHz system clock frequency**
  - Most of the peripherals of the MiniRISC system are available in the simulator
- **Run on the hardware**
  - Available only, if the FPGA board is connected to the PC

MiniRISC IDE

- **Compiling the program: *Compile* (, F5) button**
  - The error messages appear in the console
  - The wrong lines are underlined with red color
- **Downloading the compiled program: *Download* (, F6) btn.**
  - If the FPGA hasn't configured yet, the MiniRISC system is downloaded first
  - The execution stops at address 0x00 (reset vector)
- **Controlling the execution of the program**
  - ***Run*** (▶, F7): resumes the execution of the program
  - ***Break*** (⏸, F8): suspends the execution of the program, the next instruction is highlighted with yellow color
  - ***Step*** (, F10): the current instruction is executed and the execution stops at the next instruction

MiniRISC IDE

- **Controlling the execution of the program**
  - ***Auto step*** (☰): the ***Step*** command issued automatically
    - The frequency can be set in the Debug menu
    - The auto stepping can be stopped using the ***Break*** command
  - ***Reset*** (↰, F9): issues a reset signal to the processor system
  - ***Stop*** (■): stops the execution of the program
    - After the command, the program has to be downloaded again
- **Breakpoints can be placed to any assembly instruction in the source code editor by clicking on the margin**
  - The breakpoint is indicated with a red circle on the margin

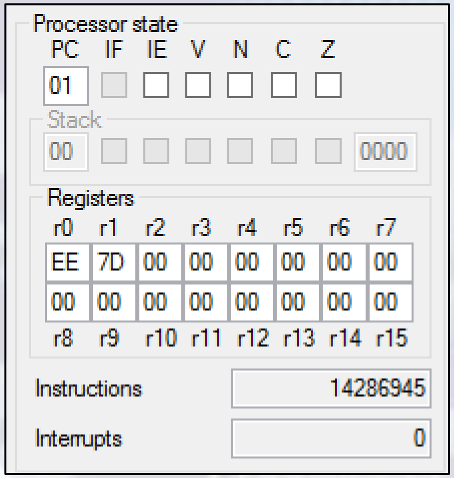


```
10      mov     r0, SW      ; A kapcsolók állapota r0-ba kerül.
11 ●    mov     LD, r0      ; A LED-ekre kiírjuk r0 értékét.
12      jmp     start       ; Ugrás a ciklus elejére.
```

  - Stepping to a breakpoint suspends the execution
    - Basically, a hardware ***Break*** command is issued

MiniRISC IDE

- **Modifying the processor state and the data memory, and controlling the peripherals is possible only when the execution of the program is suspended (break state)**
- ***Processor state*** **panel**
  - Value of the program counter (PC)
  - Value of the flags (Z, C N, V, IE, IF – read-only)
  - Value of the top of the stack (read-only)
  - Value of the registers
  - Number of the executed instructions
  - Number of the accepted interrupt requests
- **Control panel of the basic peripherals**
  - Displays the state of the LEDs, the DIP switches and the push buttons
  - Allows register-level control



Processor state

PC	IF	IE	V	N	C	Z
01	☐	☐	☐	☐	☐	☐

Stack

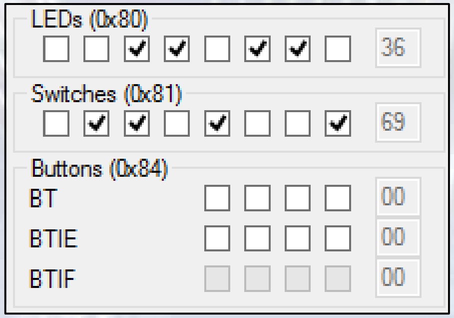
00 ☐ ☐ ☐ ☐ ☐ ☐ 0000

Registers

r0	r1	r2	r3	r4	r5	r6	r7
EE	7D	00	00	00	00	00	00
00	00	00	00	00	00	00	00
r8	r9	r10	r11	r12	r13	r14	r15

Instructions 14286945

Interrupts 0



LEDs (0x80)

☐ ☐ ☑ ☑ ☐ ☑ ☑ ☐ 36

Switches (0x81)

☐ ☑ ☑ ☐ ☑ ☐ ☐ ☑ 69

Buttons (0x84)

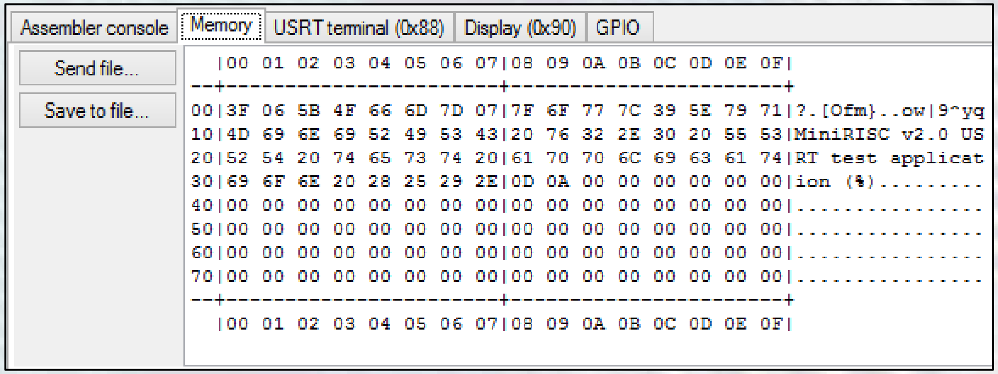
BT ☐ ☐ ☐ ☐ 00

BTIE ☐ ☐ ☐ ☐ 00

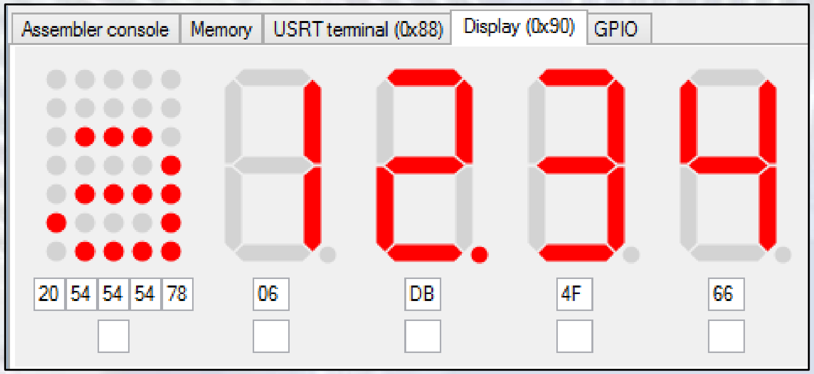
BTIF ☐ ☐ ☐ ☐ 00

MiniRISC IDE

- ***Memory*** **window**
  - Displays the content of the 128 x 8 bit data memory
  - Each byte can be modified by clicking on them
  - The memory content can be loaded from file (***Send file…*** button) and can be saved to file (***Save to file…*** button)
- ***Display*** **window**
  - Allows controlling the seven-segment and the dot-matrix displays
  - The display segments can be turned on/off by clicking on them
  - The segment values and the character to be displayed can be specified in the textboxes



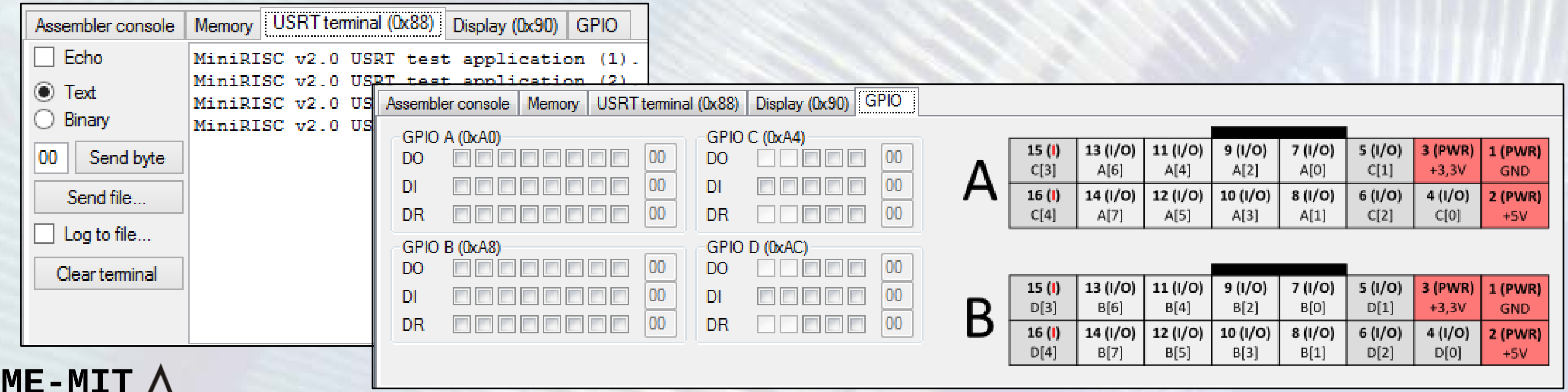
```
  |00 01 02 03 04 05 06 07|08 09 0A 0B 0C 0D 0E 0F|
--+-----------------------+-----------------------+
00|3F 06 5B 4F 66 6D 7D 07|7F 6F 77 7C 39 5E 79 71|?.[Ofm}..ow|9^yq
10|4D 69 6E 69 52 49 53 43|20 76 32 2E 30 20 55 53|MiniRISC v2.0 US
20|52 54 20 74 65 73 74 20|61 70 70 6C 69 63 61 74|RT test applicat
30|69 6F 6E 20 28 25 29 2E|0D 0A 00 00 00 00 00 00|ion (%).........
40|00 00 00 00 00 00 00 00|00 00 00 00 00 00 00 00|................
50|00 00 00 00 00 00 00 00|00 00 00 00 00 00 00 00|................
60|00 00 00 00 00 00 00 00|00 00 00 00 00 00 00 00|................
70|00 00 00 00 00 00 00 00|00 00 00 00 00 00 00 00|................
--+-----------------------+-----------------------+
  |00 01 02 03 04 05 06 07|08 09 0A 0B 0C 0D 0E 0F|
```



20 54 54 54 78 | 06 | DB | 4F | 66

MiniRISC IDE

- ***USRT terminal*** **window**
  - Provides serial communication with the MiniRISC system
  - The pressed characters will be sent, the received characters are displayed in the terminal window
  - Files can be sent, received data can be saved to a file
- **GPIO window**
  - Displays the state of the GPIO peripherals and provides their register-level control (output data, input data, direction)
  - Pinout of the expansion connectors



☐ Echo
● Text
○ Binary
00
☐ Log to file...

```
MiniRISC v2.0 USRT test application (1).
MiniRISC v2.0 USRT test application (2)
MiniRISC v2.0 US
MiniRISC v2.0 US
```

GPIO A (0xA0)
DO ☐☐☐☐☐☐☐☐ 00
DI ☐☐☐☐☐☐☐☐ 00
DR ☐☐☐☐☐☐☐☐ 00

GPIO B (0xA8)
DO ☐☐☐☐☐☐☐☐ 00
DI ☐☐☐☐☐☐☐☐ 00
DR ☐☐☐☐☐☐☐☐ 00

GPIO C (0xA4)
DO ☐☐☐☐☐ 00
DI ☐☐☐☐☐ 00
DR ☐☐☐☐☐ 00

GPIO D (0xAC)
DO ☐☐☐☐☐ 00
DI ☐☐☐☐☐ 00
DR ☐☐☐☐☐ 00

A

15 (I) C[3]	13 (I/O) A[6]	11 (I/O) A[4]	9 (I/O) A[2]	7 (I/O) A[0]	5 (I/O) C[1]	3 (PWR) +3,3V	1 (PWR) GND
16 (I) C[4]	14 (I/O) A[7]	12 (I/O) A[5]	10 (I/O) A[3]	8 (I/O) A[1]	6 (I/O) C[2]	4 (I/O) C[0]	2 (PWR) +5V

B

15 (I) D[3]	13 (I/O) B[6]	11 (I/O) B[4]	9 (I/O) B[2]	7 (I/O) B[0]	5 (I/O) D[1]	3 (PWR) +3,3V	1 (PWR) GND
16 (I) D[4]	14 (I/O) B[7]	12 (I/O) B[5]	10 (I/O) B[3]	8 (I/O) B[1]	6 (I/O) D[2]	4 (I/O) D[0]	2 (PWR) +5V

Steps of the software development

- Think over the problem and create the basic concept of the solution
- Decompose the problem into MiniRISC assembly instructions and create the source code
- The source code can be compiled using the ***Compile (F5)*** command
- If there are errors, correct them
- If the program compiled without errors, it can be downloaded using the ***Download (F6)*** command
- Verify the operation of the program in debug mode using the services of the MiniRISC IDE

Example programs

Example 1: displaying the state of the DIP switch on the LEDs

- Very simple: in an endless loop, read the state of the DIP switch (0x81) and write it to the LEDs (0x81)

```
DEF LD   0x80                 ; LED register
DEF SW   0x81                 ; DIP switch register

    CODE

;*************************************************************
;
;* Start of the program. The addr. 0x00 is the reset vector. *
;*************************************************************
;
start:
    mov      r0, SW           ; Read the state of the sw. to r0.
    mov      LD, r0           ; Value of r0 is written to the LEDs.
    jmp      start            ; Jump to the beginning of the loop.
```

Example programs

Example 1: displaying the state of the DIP switch on the LEDs

- Content of the list file generated by the assembler

```
LOGSYS MiniRISC v2.0 assembler v1.0 list file
Copyright (C) 2013 LOGSYS, Tamas Raikovich
Source file: example1.s
Created on : 2013.03.27. 12:51:08

S Addr  Instr   Source code
--------------------------------------------------------------------------------
                DEF LD 0x80                 ; LED register
                DEF SW 0x81                 ; DIP switch register

                    CODE

                ;*************************************************************
                ;* Start of the program. The addr. 0x00 is the reset vector. *
                ;*************************************************************
C 00            start:
C 00    D081        mov     r0, SW[81]      ; Read the state of the sw. to r0.
C 01    9080        mov     LD[80], r0      ; Value of r0 is written to the LEDs.
C 02    B000        jmp     start[00]       ; Jump to the beginning of the loop.
```