



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



MiniRISC processor, 2013.05.08. (v1.0)

FPGA labor

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)



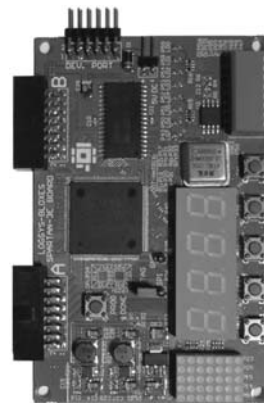
MiniRISC processor, 2013.05.08. (v1.0)

1

FPGA labor

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, 2013.05.08. (v1.0)

2

FPGA labor

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**



MiniRISC processor, 2013.05.08. (v1.0)

3

FPGA labor

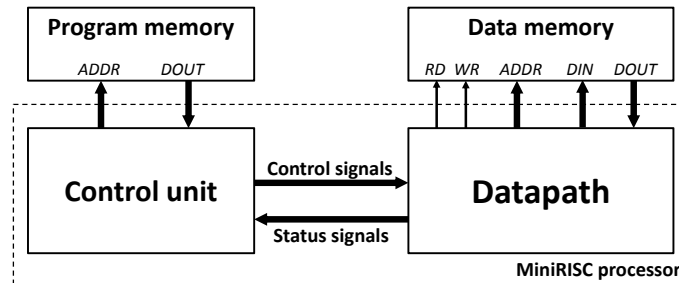
Contents

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 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



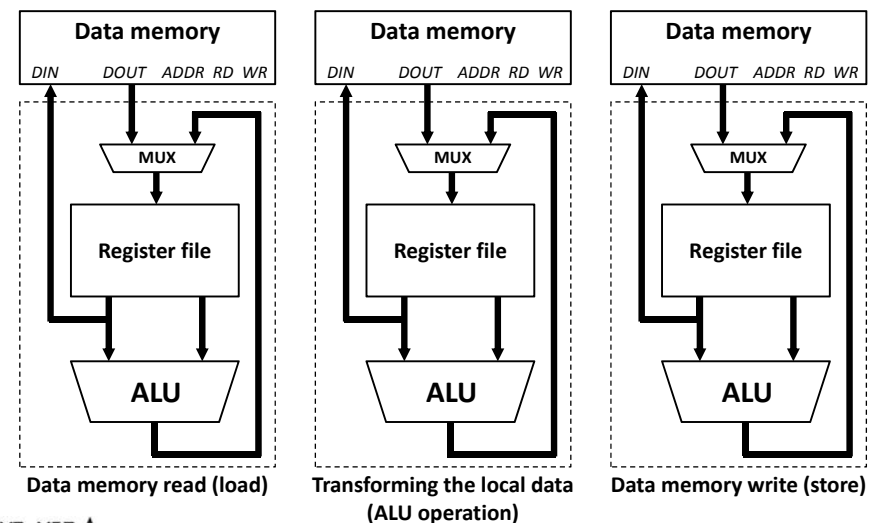
MiniRISC processor - Structure

(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

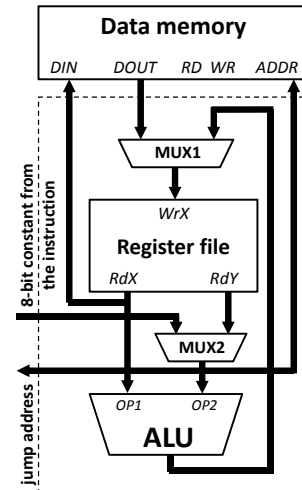
(Datapath)



MiniRISC processor - Structure

(Datapath of the MiniRISC processor)

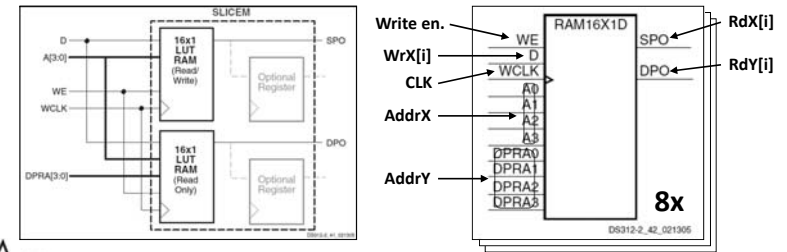
- Extended version of the basic datapath
- Write data select multiplexer for the register file (MUX1)
 - ALU result or data memory
- 16 x 8 bit register file
 - Two register addresses are used
- Arithmetic-logic unit (ALU)
 - Register
 - 8-bit constant from the instruction
- Selecting the 2nd ALU operand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

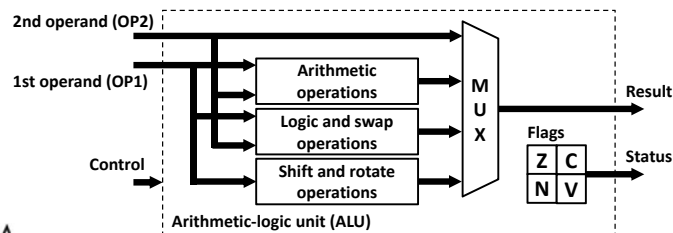
- The register file is implemented using distributed RAM (FPGA resource)
- Distributed RAM can be used to store small amount of data efficiently
 - 1 write port and 1 or 2 read ports
 - The write and the first read port has shared address input ($A=AddrX$)
 - The address for the second read port can 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)

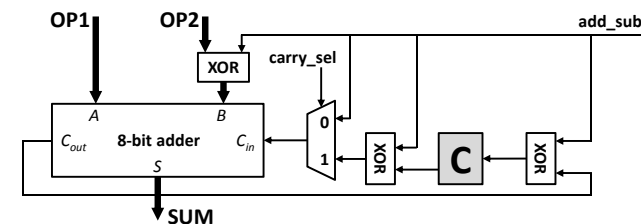
- The ALU executes different operations on the local data
 - Data moving: no operation, the result is OP2
 - Arithmetic: addition and subtraction with or without carry
 - Logic: bitwise AND, OR, XOR
 - Shift, rotate and swap
- Status flags give information about the result of the operations
 - Zero (Z), 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: $\{C_{out}, SUM\} = OP1 + \overline{OP2} + 1$
 - Subtraction with borrow: $\{C_{out}, SUM\} = OP1 + \overline{OP2} + \overline{C_{in}}$



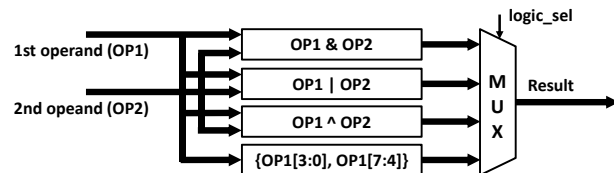
MiniRISC processor - Structure

(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



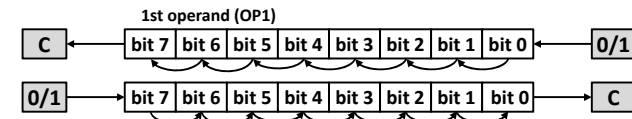
- Implemented in the logic operation block
 - The 4th input of the MUX can be used for the swap operation
 - Same status flags are modified (Z and N) → same control



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



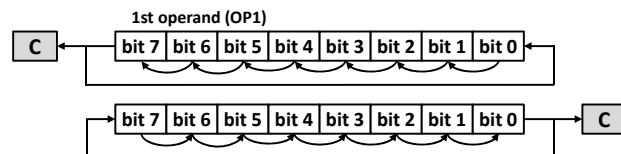
- **Arithmetic shift right**
 - When a signed number is shifted right, the value of the sign bit (MSb) should be preserved in order to get correct result
 - The shifted out bit is stored in the carry (C) flag
 - There is no separate arithmetic shift left operation because it is the same as the logical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



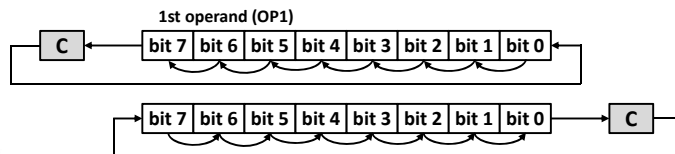
MiniRISC processor - Structure

(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

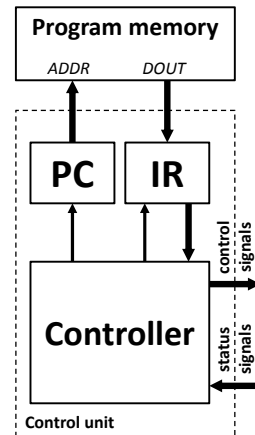
(Datapath of the MiniRISC processor – ALU)

- **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 $C_{in7} \text{ xor } C_{out7}$ method cannot be used because the carry in bit of the MSb is not available inside the FPGA)

MiniRISC processor - Structure

(Control unit)

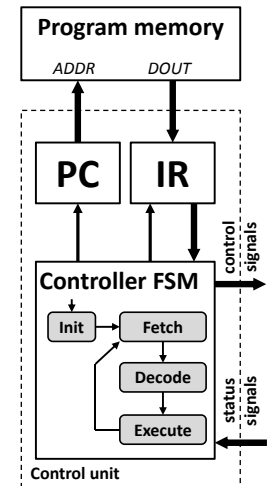
- **Example:** $DMEM[3] = DMEM[0] + DMEM[1]$, this requires 4 datapath operations:
 1. $REG[0] = DMEM[0]$ (load)
 2. $REG[1] = DMEM[1]$ (load)
 3. $REG[1] = REG[0] + REG[1]$ (ALU operation)
 4. $DMEM[3] = REG[1]$ (store)
- **Instruction:** operation that the CPU can execute
- **Program:** series of instructions
 - 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

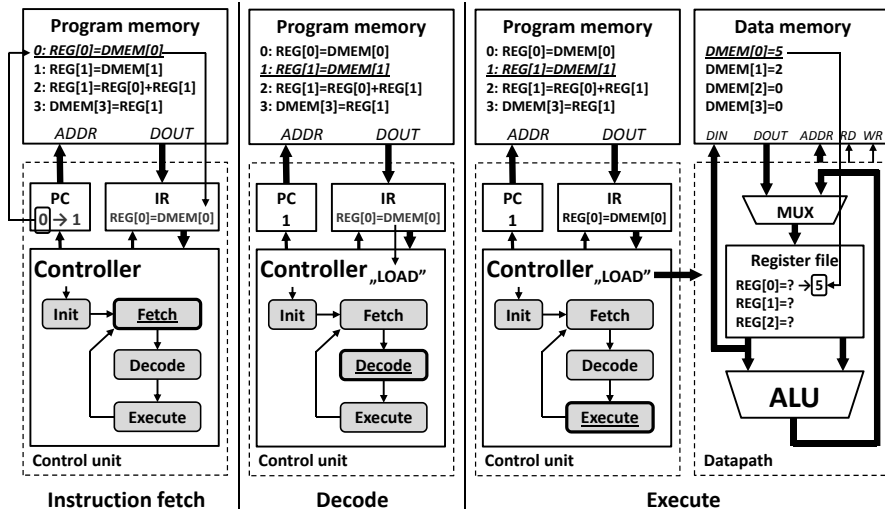
(Control unit)

- **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



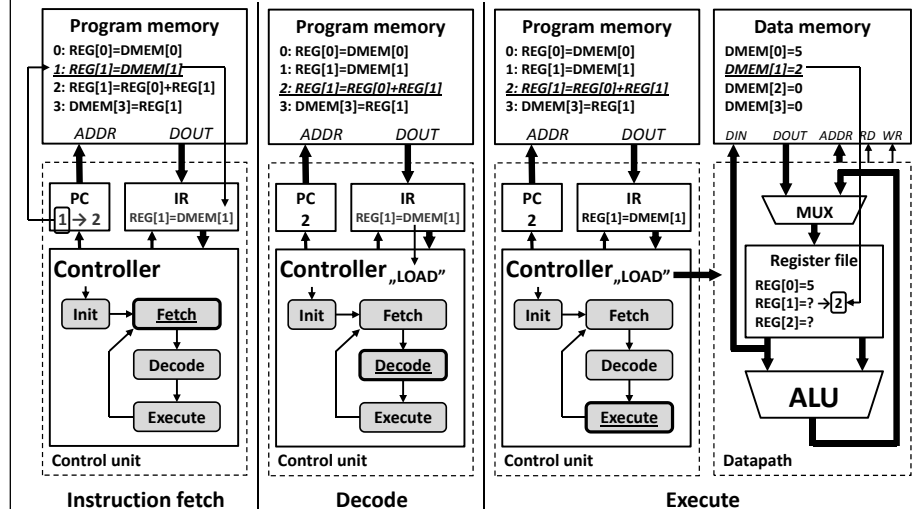
MiniRISC processor - Structure

(Control unit)



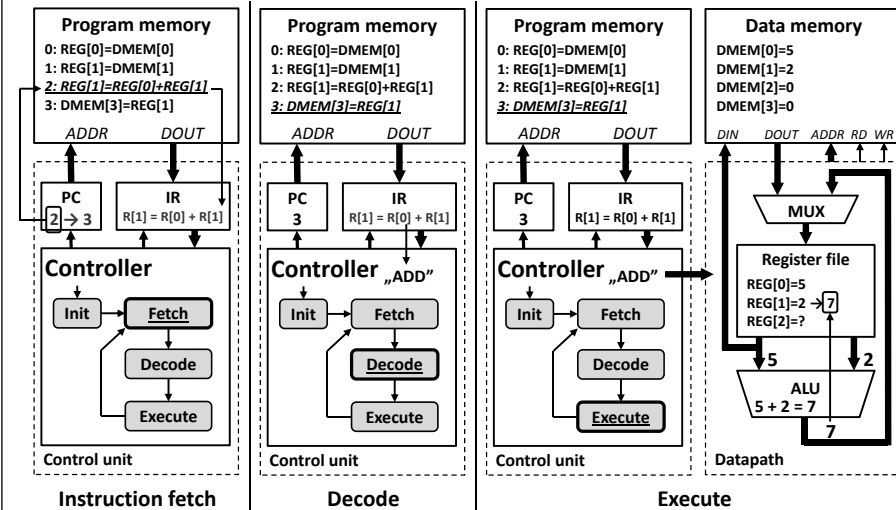
MiniRISC processor - Structure

(Control unit)



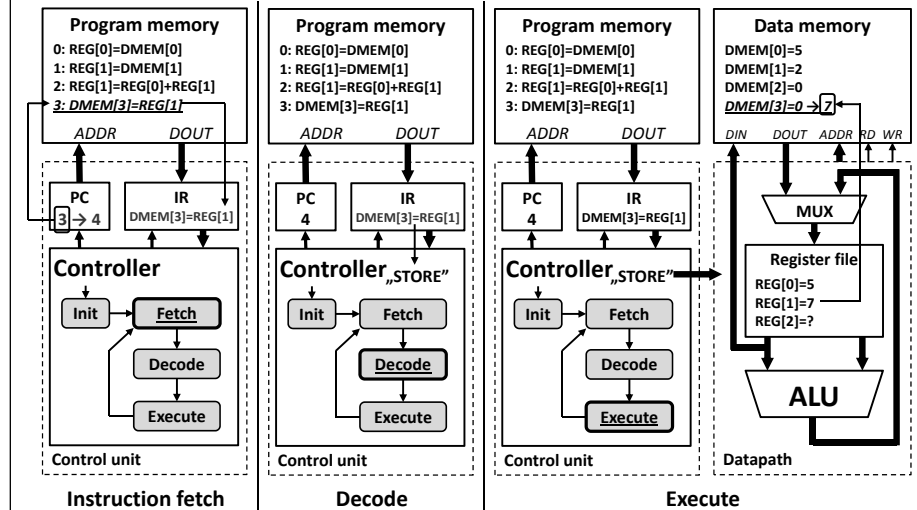
MiniRISC processor - Structure

(Control unit)



MiniRISC processor - Structure

(Control unit)



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**
 - **Opcod:** determines the operation to be executed
 - **Operands:** data used by the given operation
 - Register
 - Constant value

MiniRISC processor - Structure

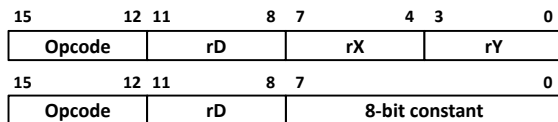
(Control unit of the MiniRISC CPU – Instruction set)

- 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)

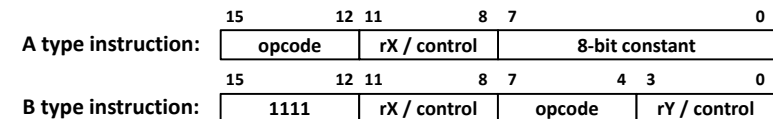
- How many operands should be in the 16-bit instructions?
- Separate addresses for each port of the register file
 - The program contains less number of instructions
 - The instructions are wider
 - Operands: 12 bits
 - Two source register (rX, rY) and a destination register (rD)
 - One register (rD)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



MiniRISC processor - Structure

(Control unit of the MiniRISC CPU – Instruction set)

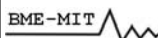
- How many operands should be in the 16-bit instructions?
- Two register addresses
 - Good tradeoff between the number of instructions and the length of instructions
 - Two main instruction type according to the operands
 - A register (rX) and an 8-bit constant → 12 bits A type
 - Two registers (rX and 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**



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



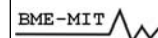
MiniRISC processor - Structure

(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	$rX \leftarrow \text{DMEM}[\text{addr}]$	-	-	-	-
1111xxxx1101yyyy	MOV rX, (rY)	$rX \leftarrow \text{DMEM}[rY]$	-	-	-	-
1001xxxxaaaaaaaa	MOV addr, rX	$\text{DMEM}[\text{addr}] \leftarrow rX$	-	-	-	-
1111xxxx1001yyyy	MOV (rY), rX	$\text{DMEM}[rY] \leftarrow rX$	-	-	-	-
1100xxxxiiiiiiii	MOV rX, #imm	$rX \leftarrow \text{imm}$	-	-	-	-
1111xxxx1100yyyy	MOV rX, rY	$rX \leftarrow rY$	-	-	-	-



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
0000xxxxiiiiiii	ADD rX, #imm	$rX \leftarrow rX + \text{imm}$	+	+	+	+
0001xxxxiiiiiii	ADC rX, #imm	$rX \leftarrow rX + \text{imm} + C$	+	+	+	+
0010xxxxiiiiiii	SUB rX, #imm	$rX \leftarrow rX - \text{imm}$	+	+	+	+
0011xxxxiiiiiii	SBC rX, #imm	$rX \leftarrow rX - \text{imm} - C$	+	+	+	+
1010xxxxiiiiiii	CMP rX, #imm	$rX - \text{imm}$	+	+	+	+

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	$rX \leftarrow rX + rY$	+	+	+	+
1111xxxx0001yyyy	ADC rX, rY	$rX \leftarrow rX + rY + C$	+	+	+	+
1111xxxx0010yyyy	SUB rX, rY	$rX \leftarrow rX - rY$	+	+	+	+
1111xxxx0011yyyy	SBC rX, rY	$rX \leftarrow rX - rY - C$	+	+	+	+
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

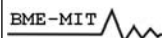
(Control unit of the MiniRISC CPU – Instruction set)

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
0100xxxxiiiiiii	AND rX, #imm	$rX \leftarrow rX \& \text{imm}$	+	-	+	-
0101xxxxiiiiiii	OR rX, #imm	$rX \leftarrow rX \mid \text{imm}$	+	-	+	-
0110xxxxiiiiiii	XOR rX, #imm	$rX \leftarrow rX \wedge \text{imm}$	+	-	+	-
0111xxxx00000000	SWP rX	$rX \leftarrow \{rX[3:0], rX[7:4]\}$	+	-	+	-
1000xxxxiiiiiii	TST rX, #imm	$rX \& \text{imm}$	+	-	+	-

Operation select (00: AND, 01: OR, 10: XOR, 11: swap)
 1 if the result of the logic operation is not stored



MiniRISC processor - Structure

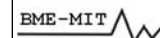
(Control unit of the MiniRISC CPU – Instruction set)

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	$rX \leftarrow rX \& rY$	+	-	+	-
1111xxxx0101yyyy	OR rX, rY	$rX \leftarrow rX \mid rY$	+	-	+	-
1111xxxx0110yyyy	XOR rX, rY	$rX \leftarrow rX \wedge rY$	+	-	+	-
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	$rX \leftarrow \{rX[6:0], 0\}$	+	+	+	-
1111xxxx01110100	SL1 rX	$rX \leftarrow \{rX[6:0], 1\}$	+	+	+	-
1111xxxx01110001	SR0 rX	$rX \leftarrow \{0, rX[7:1]\}$	+	+	+	-
1111xxxx01110101	SR1 rX	$rX \leftarrow \{1, rX[7:1]\}$	+	+	+	-
1111xxxx01111001	ASR rX	$rX \leftarrow \{rX[7], rX[7:1]\}$	+	+	+	-

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	$rX \leftarrow \{rX[6:0], rX[7]\}$	+	+	+	-
1111xxxx01110011	ROR rX	$rX \leftarrow \{rX[0], rX[7:1]\}$	+	+	+	-
1111xxxx01110110	RLC rX	$rX \leftarrow \{rX[6:0], C\}$	+	+	+	-
1111xxxx01110111	RRC rX	$rX \leftarrow \{C, rX[7:1]\}$	+	+	+	-

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	$PC \leftarrow \text{addr}$	-	-	-	-
111100001011yyyy	JMP (rY)	$PC \leftarrow rY$	-	-	-	-
10110001aaaaaaaa	JZ addr	$PC \leftarrow \text{addr}, \text{ if } Z=1$	-	-	-	-
111100011011yyyy	JZ (rY)	$PC \leftarrow rY, \text{ if } Z=1$	-	-	-	-
10110010aaaaaaaa	JNZ addr	$PC \leftarrow \text{addr}, \text{ if } Z=0$	-	-	-	-
111100101011yyyy	JNZ (rY)	$PC \leftarrow rY, \text{ if } Z=0$	-	-	-	-



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	$PC \leftarrow \text{addr}, \text{ if } C=1$	-	-	-	-
111100111011yyyy	JC (rY)	$PC \leftarrow rY, \text{ if } C=1$	-	-	-	-
10110100aaaaaaaa	JNC addr	$PC \leftarrow \text{addr}, \text{ if } C=0$	-	-	-	-
111101001011yyyy	JNC (rY)	$PC \leftarrow rY, \text{ if } C=0$	-	-	-	-
10110101aaaaaaaa	JN addr	$PC \leftarrow \text{addr}, \text{ if } N=1$	-	-	-	-
111101011011yyyy	JN (rY)	$PC \leftarrow rY, \text{ if } N=1$	-	-	-	-



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	$PC \leftarrow \text{addr}$, if $N=0$	-	-	-	-
111101101011yyyy	JNN (rY)	$PC \leftarrow rY$, if $N=0$	-	-	-	-
10110111aaaaaaaa	JV addr	$PC \leftarrow \text{addr}$, if $V=1$	-	-	-	-
111101111011yyyy	JV (rY)	$PC \leftarrow rY$, if $V=1$	-	-	-	-
10111000aaaaaaaa	JNV addr	$PC \leftarrow \text{addr}$, if $V=0$	-	-	-	-
111110001011yyyy	JNV (rY)	$PC \leftarrow rY$, if $V=0$	-	-	-	-



MiniRISC processor - Structure

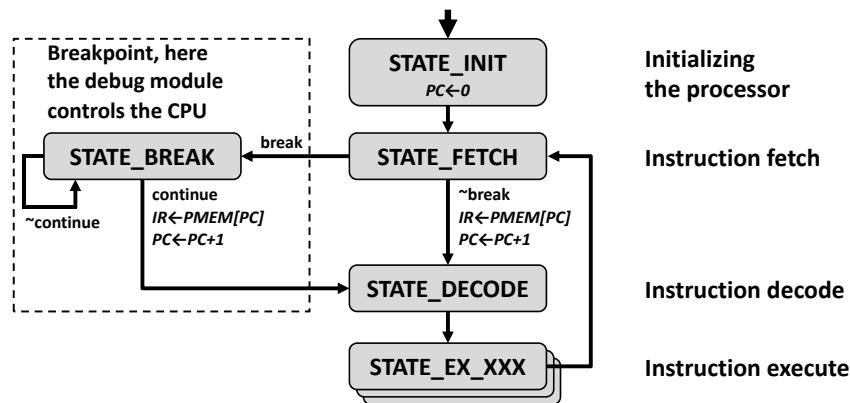
(Control unit of the MiniRISC CPU – IR and PC)

- **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 0x00
 - Jump: loaded with the jump address
 - 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)



MiniRISC processor - Structure

(Control unit of the MiniRISC CPU – Controller FSM)

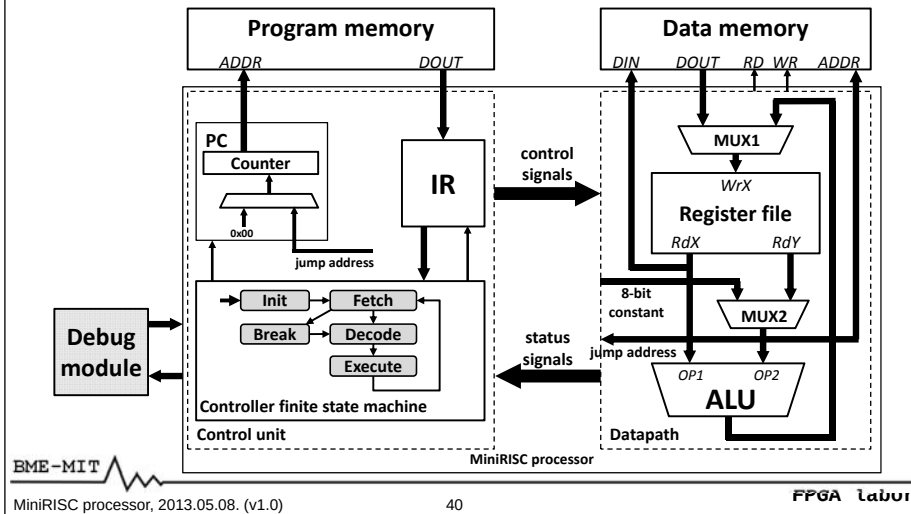
- **Breakpoint state (STATE_BREAK)**
 - Serves software development supporting purposes
 - The debug module takes over the control (details later)
 - **Break** signal in the fetch phase → execution stops, BREAK state
 - **Continue** signal in the BREAK state → execution resumes
- **Instruction execute**
 - 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)

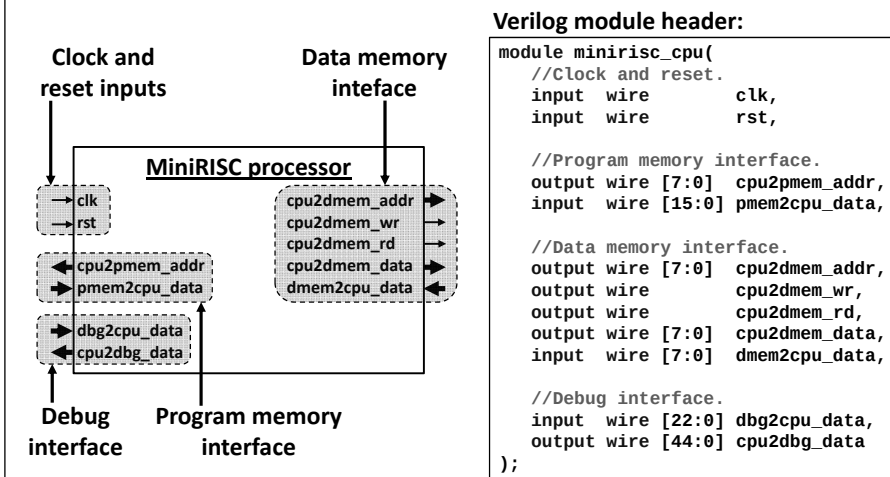
The detailed structure can be found in the Verilog source code



Contents

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)

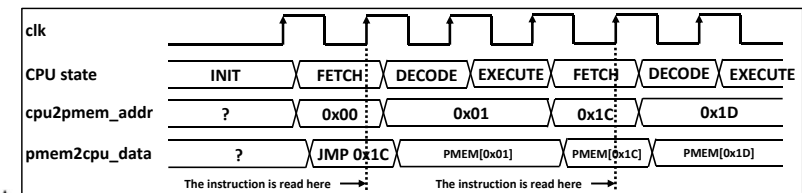
MiniRISC processor – Interfaces



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

(Data memory interface)

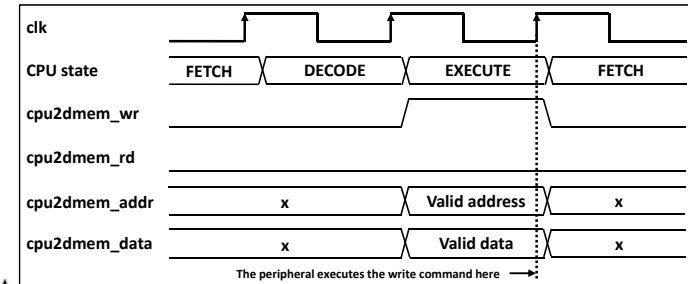
- **Data memory interface**
 - Simple synchronous bus, commands are valid at the rising clock edge
 - **cpu2dmem_addr**: 8-bit address bus
 - **cpu2dmem_wr**: active-high write enable signal
 - **cpu2dmem_rd**: active-high read enable signal
 - **cpu2dmem_data**: 8-bit write data bus (CPU → peripheral)
 - **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 damage 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)

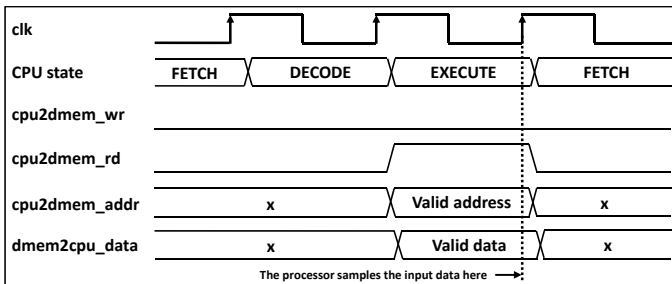
- **Write cycle of the data memory interface**
 - In the execute phase, the write cycle is indicated by the **cpu2dmem_wr** signal which is active for 1 clock cycle
 - During the write cycle, the **cpu2dmem_addr** address is stable
 - During the write cycle, the **cpu2dmem_data** data is stable, which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



MiniRISC processor – Interfaces

(Data memory interface – Read cycle)

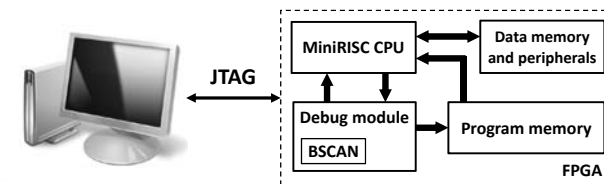
- **Read cycle of the data memory interface**
 - In the execute phase, the read cycle is indicated by the **cpu2dmem_rd** signal which is active for 1 clock cycle
 - During the read cycle, the **cpu2dmem_addr** address is stable
 - During the read cycle, the selected peripheral drives the **dmem2cpu_data**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



MiniRISC processor – Interfaces

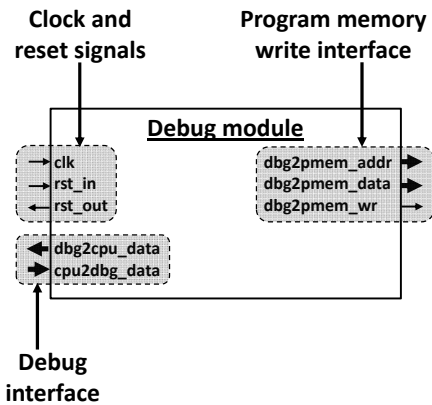
(Debug module, debug interface)

- **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**
 - $psel = ((cpu2dmem_addr \gg N) == (BASEADDR \gg N))$
 - 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)

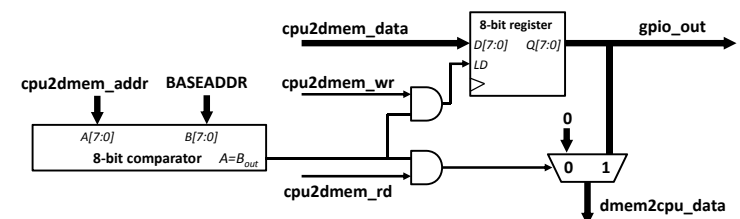
Checking the lower address bits is required, if $N > 0$

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 $\rightarrow$ 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_out <= cpu2dmem_data;

//Driving the read data bus.
always @(*)
    if (dreg_rd)
        dmem2cpu_data <= gpio_out;
    else
        dmem2cpu_data <= 8'd0;
endmodule
```

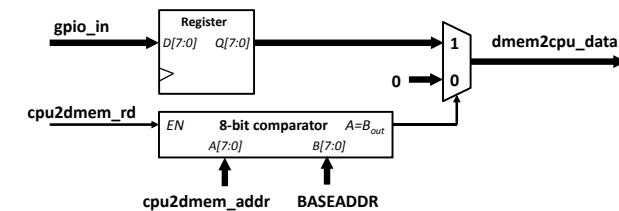


MiniRISC processor – I/O extension

(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
);

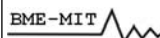
//Select signal of the peripheral.
wire psel = (cpu2dmem_addr == BASEADDR);

//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
```

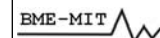
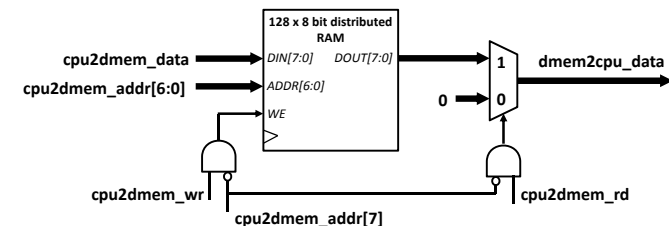


MiniRISC processor – I/O extension

(Examples)

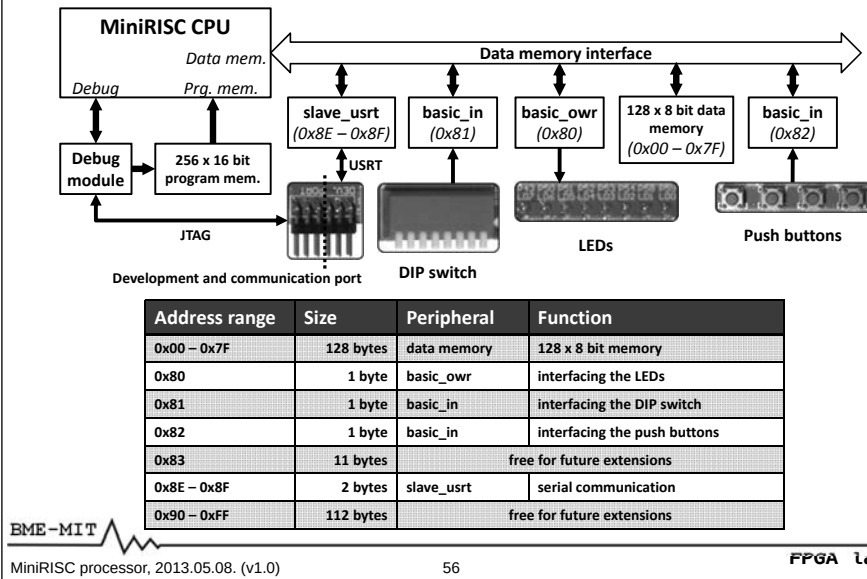
Example 3: 128 x 8 bit data memory

- 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

(Block diagram)



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 – slave_usrt)

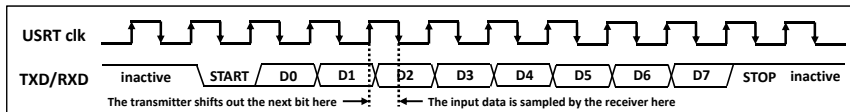
slave_usrt: peripheral that provides serial communication

- 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

- Control and status register**

BASEADDR + 0x00, writable and readable

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
RXNE	TXNF	0	0	0	0	RXEN	TXEN
R	R	R	R	R	R	R/W	R/W



MiniRISC system

(Peripherals – slave_usrt)

slave_usrt: peripheral that provides serial communication

- Control and status register**

Bit	Mode	Function
TXEN	R/W	0: the USRT transmitter is disabled 1: the USRT transmitter is enabled
RXEN	R/W	0: the USRT receiver is disabled 1: the USRT receiver is enabled
TXNF	R	0: the transmit data register is full, it cannot be written 1: the transmit data register is empty
RXNE	R	0: the receive data register is empty, it cannot be read 1: the receive data register contains valid data

- Data register: BASEADDR + 0x01, writable and readable**
 - Write: data to send (can be written if TXNF=1)
 - Read: received data (can be read 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

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
- 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
 - *filename.code.hex* text file for initializing the program memory from the Verilog source code
 - *filename.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

- Source 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: '\', '\'', '\n', '\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

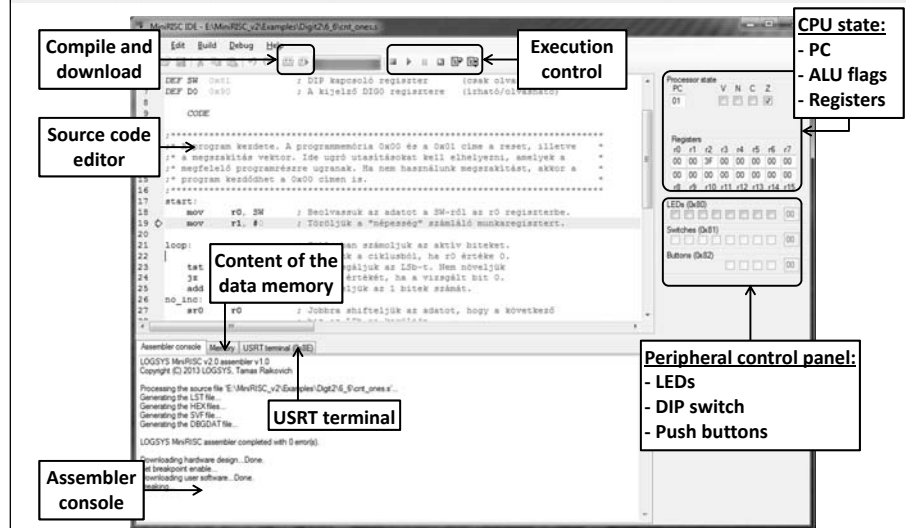


MiniRISC assembler

- 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)



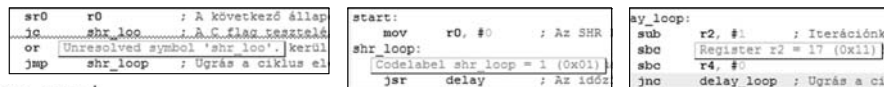
MiniRISC IDE



MiniRISC IDE

Capabilities of the source code editor:

- **Syntax highlighting**
- **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 állapot
lc     shr_loo ; A C flag tesztelés
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)
jnr    delay ; Az idős

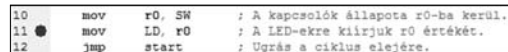
ay_loop:
sub    r2, #1 ; Iterációknk
sbc    Register r2 = 17 (0x11)
sbc    r4, #
jnc    delay_loop ; Ugrás a ci
    
```

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

- **Processor state panel**
 - Displays the state of the processor:
 - Value of the program counter (PC)
 - Value of the flags (Z, C, N, V)
 - Value of the CPU registers
 - Each item can be modified
- **Peripheral control panel**
 - Displays the state of the LEDs, the DIP switches and the push buttons
 - Allows controlling these peripherals

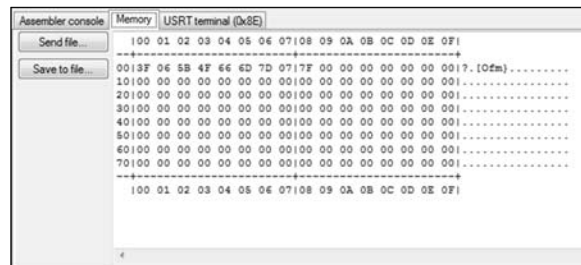
Processor state									
PC		V	N	C	Z				
02		☐	☐	☐	☒				
Registers									
r0	r1	r2	r3	r4	r5	r6	r7		
00	00	3F	00	00	00	00	00		
00	00	00	00	00	00	00	00		
r8	r9	r10	r11	r12	r13	r14	r15		

LEDs (0x80)												
0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	4	5	6	7	8	9	10	11	12
Switches (0x81)												
0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	4	5	6	7	8	9	10	11	12
Buttons (0x82)												
0	1	2	3	4	5	6	7	8	9	10	11	12

MiniRISC IDE

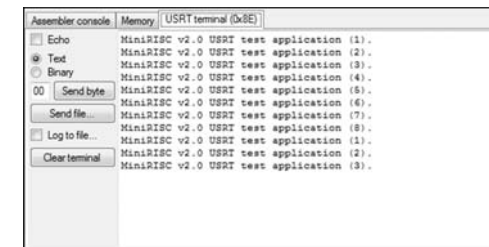
• Memory window

- 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 and can be saved to file



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**
- **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



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.