FB Elektrotechnik und Informationstechnik
Prof. Dr.-Ing. Norbert Wehn
Dozent:
Uwe Wasenmüller
Raum 12-213, wa@eit.uni-kl.de

Task 4

Introduction

In this task of the lab as well as in task 5 to task 8 we use a typical RISC-Processor architecture *(Reduced Instruction Set Computer)*, which bases on work done by Hennessy and Patterson [HEN/PAT96] and is referred to as DLX. Its primary use is for education purposes. The DLX features the main aspects of modern commercial RISC-Processors.

The objective of this task is to teach modern concepts of processor design like *pipelining* as means for increased throughput and also to impart knowledge of the design flow of processor design. Modeling and synthesis of the DLX will be done with the hardware description language VHDL. State-of-the-art simulator ModelSim will be used to validate the models and also the mapping onto the target technology FPGA will be done using appropriate tools (Xilinx ISE).

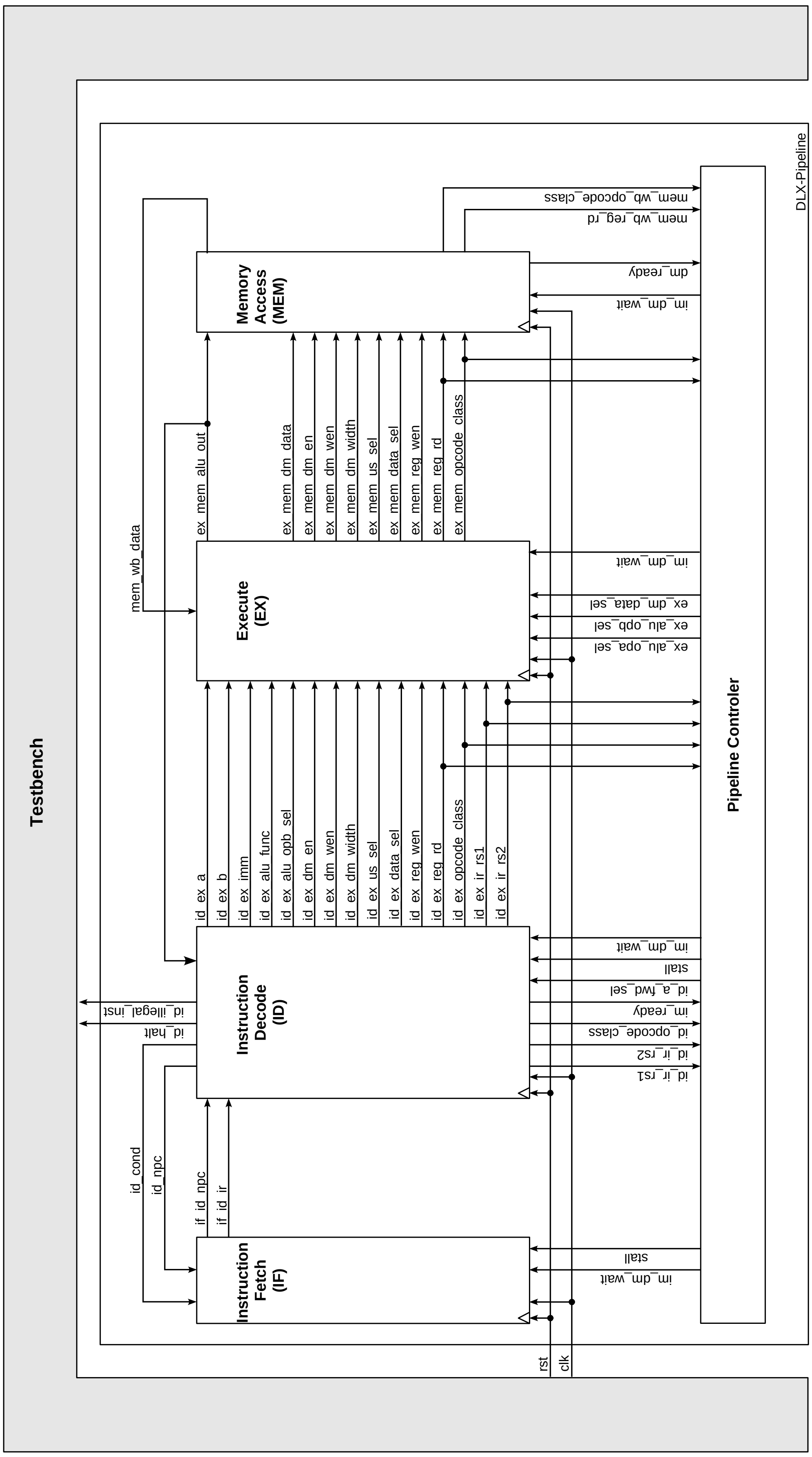
Basic knowledge in the most important language constructs of VHDL and the immanent simulation process of VHDL is mandatory for this task.

The DLX-Pipeline

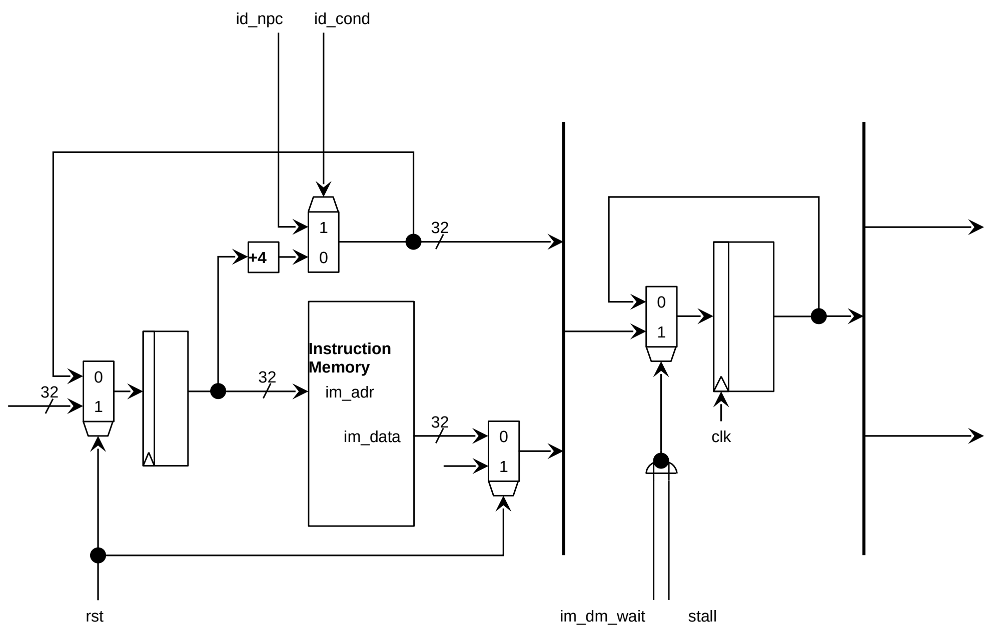
Pipelining is an implementation technique which employs overlapping execution of multiple instructions. It is a key method of the implementation of modern processor architectures.

The DLX used here uses five pipeline stages, ideally loading one instruction into the pipeline and one instruction leaves the pipeline in one clock cycle. The pipelining concept enables functional decomposition of the instruction processing over time, while the sequential positioning reflects the phases of the instruction processing.

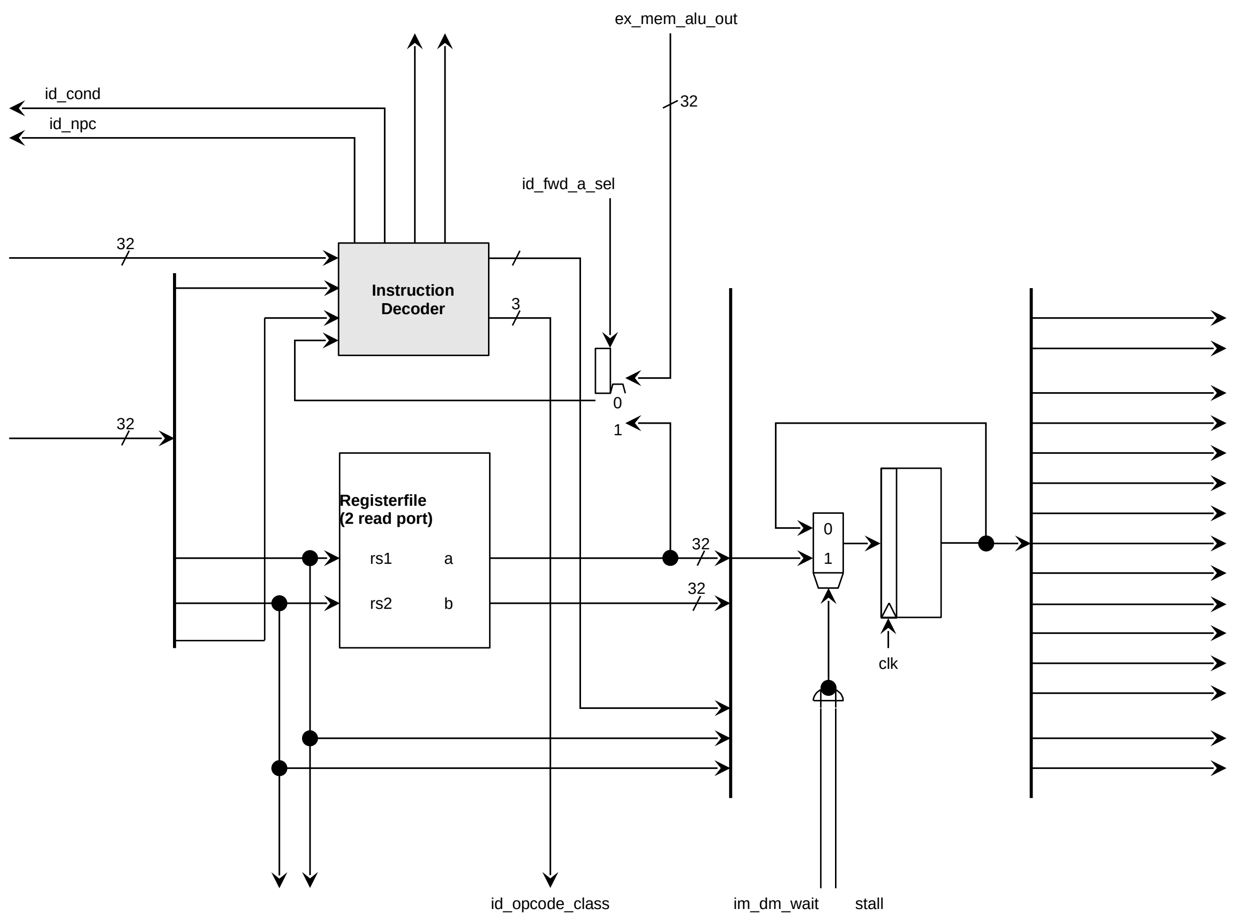
Page 2 shows a block diagram of the DLX-pipeline. Simplified schematics of the main stages of the DLX-pipeline are depicted in page 3 to page 5. Final specification is given by the VHDL codes.


Testbench
DLX-Pipeline
Pipeline Controler
Instruction Fetch (IF)
Instruction Decode (ID)
Execute (EX)
Memory Access (MEM)
rst
clk
stall
im_dm_wait
if_id_npc
if_id_ir
id_cond
id_npc
id_halt
id_illegal_inst
mem_wb_data
id_ir_rs1
id_ir_rs2
id_opcode_class
im_ready
id_a_fwd_sel
id_ex_a
id_ex_b
id_ex_imm
id_ex_alu_func
id_ex_alu_opb_sel
id_ex_dm_en
id_ex_dm_wen
id_ex_dm_width
id_ex_us_sel
id_ex_data_sel
id_ex_reg_wen
id_ex_reg_rd
id_ex_opcode_class
id_ex_ir_rs1
id_ex_ir_rs2
ex_alu_opa_sel
ex_alu_opb_sel
ex_dm_data_sel
ex_mem_alu_out
ex_mem_dm_data
ex_mem_dm_en
ex_mem_dm_wen
ex_mem_dm_width
ex_mem_us_sel
ex_mem_data_sel
ex_mem_reg_wen
ex_mem_reg_rd
ex_mem_opcode_class
dm_ready
mem_wb_reg_rd
mem_wb_opcode_class

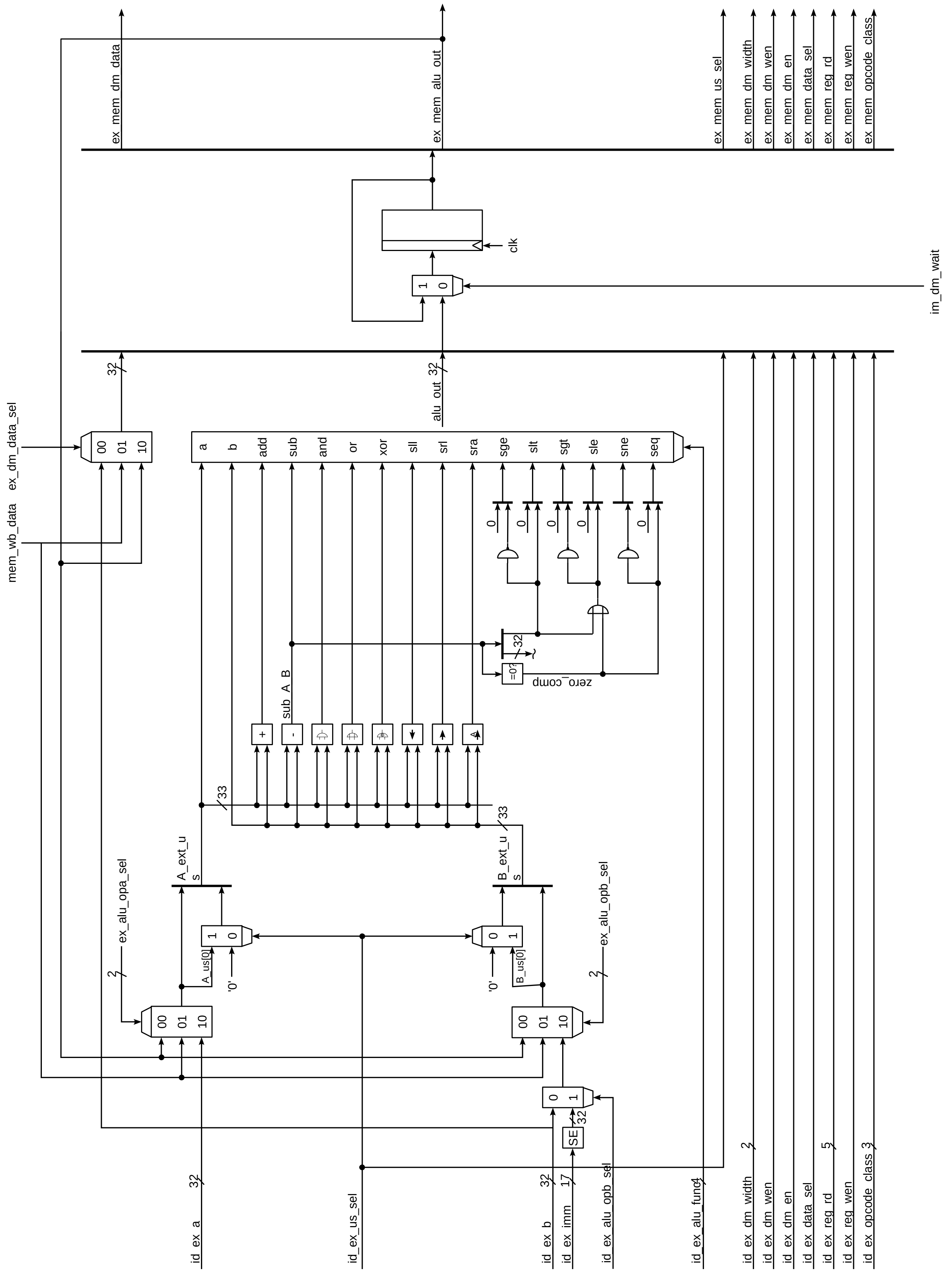
Instruction Fetch (IF)



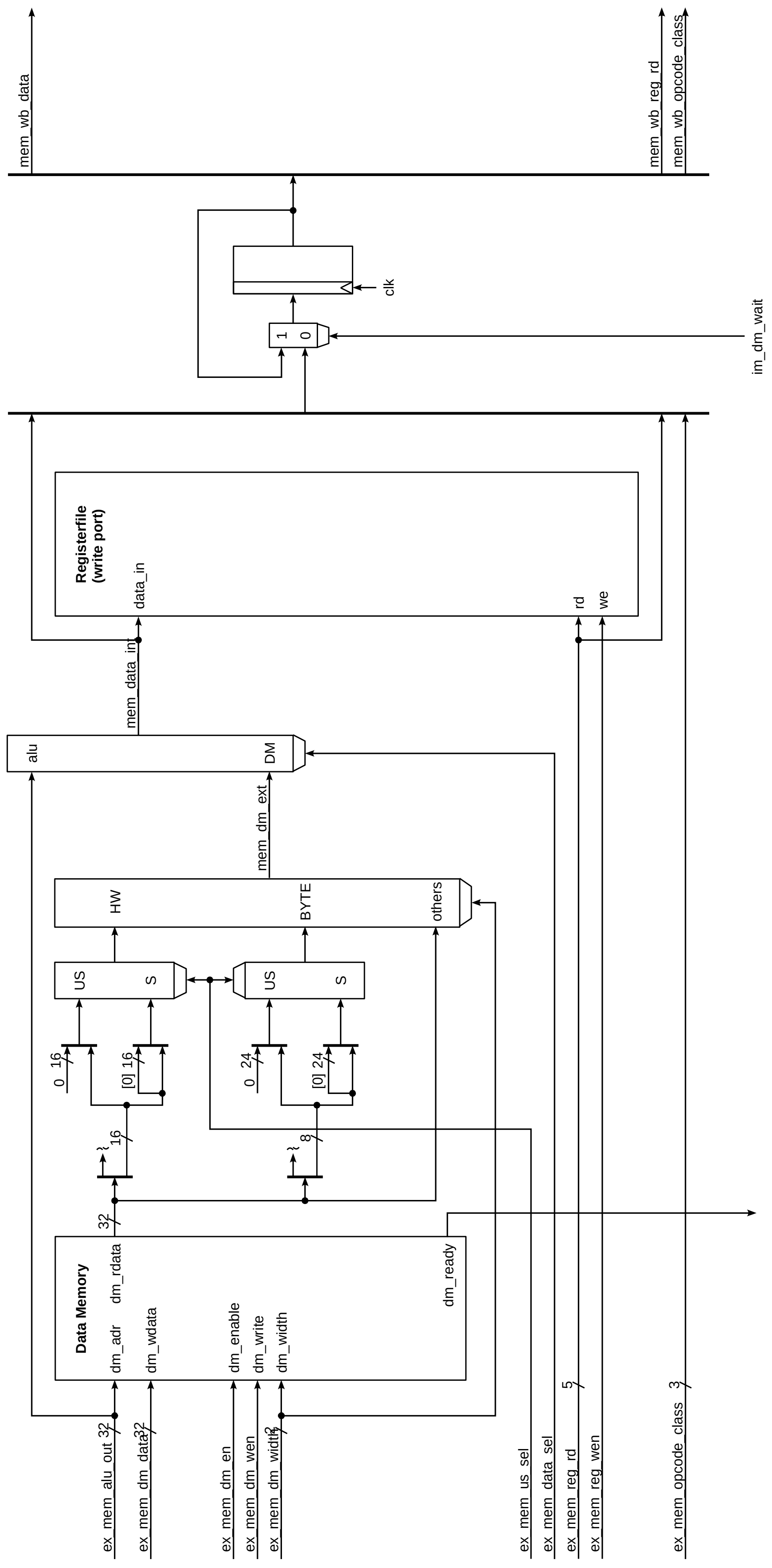
Instruction Decode (ID)



Execution (EX)



Memory Access (MEM)



Task Description

In this task the pipeline stage for instruction decoding (ID) is to be implemented in VHDL.

Open the file `dlx_pipe_id.vhd` which is relevant for this task. In this file you will find the frame for the entitity definition with the same name, which defines the input and output signals, and also the *architecture* `behaviour`, which you have to complete.

The *architecture* mainly consists of the two processes *id_comb* and *id_seq*. The first process describes all parts of the decoding logic. You have to determine the output signals in a combinatorial way, which are essentially the control signals as well as calculations for the processing of jump and branch instructions.

The second process is a clocked process, which is needed to model the necessary pipeline registers for the following stages.

Both processes show a distinction of cases (`case`), in which the different machine instructions are considered and the necessary control signals are set according to the functionality which is to be realized. Complete the case distinction for the following instructions:

ADD	- Integer Add (signed)
ADDI	- Integer Add Immediate (signed)
ADDU	- Integer Add Unsigned
ADDUI	- Integer Add Unsigned Immediate
AND	- Logical AND
ANDI	- Logical AND Immediate
BEQZ	- Branch on Integer Equal to Zero
BNEZ	- Branch on Integer Not Equal to Zero
J	- Jump
JALR	- Jump And Link Register
JAL	- Jump And Link
JR	- Jump Register
LB	- Load Byte (signed)
LBU	- Load Byte Unsigned
LH	- Load Halfword (Signed)
LHI	- Load High Immediate
LHU	- Load Halfword Unsigned
LW	- Load Word
NOP	- No Operation
OR	- Logical Or
ORI	- Logical Or Immediate (Signed)
SB	- Store Byte
SEQ	- Set On Equal To
SEQI	- Set On Equal To Immediate
SGEI	- Set On Greater or Equal To Immediate
SGE	- Set On Greater Than or Equal To
SGT	- Set On Greater Than
SGTI	- Set On Greater Than Immediate
SLE	- Set On Less Than Or Equal To
SH	- Store Halfword
SLEI	- Set On less Than or Equal To Immediate
SLL	- Shift Left Logical
SLT	- Set On Less Than
SLLI	- Shift Left Logical Immediate
SLTI	- Set On Less Than Immediate
SNE	- Set On Not Equal To
SNEI	- Set On Not Equal To Immediate

```
SRA    - Shift Right Arithmetic
SRAI   - Shift Right Arithmetic Immediate
SRL    - Shift Right Logical
SRLI   - Shift Right Logical Immediate
SUB    - Integer Subtract (signed)
SUBI   - Integer Subtract Immediate (signed)
SUBU   - Integer Subtract Unsigned
SUBUI  - Integer Subtract Unsigned Immediate
SW     - Store Word
XOR    - Logical XOR
TRAP   - Trap
XORI   - Logical Xor Immediate (Signed)
```

For testing your implementation of the ID-stage a test bench of the DLX-pipeline is provided, which uses a simple assembler test program. This test program is also shown in the appendix. Test program realizes the core functionality for prime number calculation **similar** to the well known "Sieve of Eratosthenes" algorithm. The prime numbers in the range 1 to 100 are determined by the program. The columns with hexadecimal numbers show the program address (first column) of the assembler command and the instruction word (second column) of the assembler command.

Before starting the simulation you should be familiar with the used implementation of this algorithm. The corresponding C code of the assembler instructions is given as a comment. Test bench does not verify the results; thus you must check in which format the results are given and what are the locations of the results in data memory.

Next chapter gives a concise description of the port signals of the ID-stage. Furthermore some hints for implementation are given.

Description of Port Signals

The implementation of the ID-stage requires the following input and output signals or pipeline registers:

`clk (in)`

Global clock signal, which is generated by the test bench.

`rst (in)`

Global reset signal (synchronous reset): it is set by the test bench at the start of the simulation for some clock cycles, to ensure a defined starting state of the pipeline.

`stall (in)`

Control signal of the pipeline control unit, which decides whether the IF-stage and ID-stage should stop because of a not resolvable data dependency. This case occurs especially on load operations, if the following instruction tries to read from the currently loading register. A pipeline-stall only affects the instruction currently in the ID phase and all following instructions. Previous instructions are unaffected. A stall leads to the insertion of a NOP-instruction at run time.

`dc_wait (in)`

An external control signal which is set if the command memory or the data memory cannot fulfill a read operation in the current clock cycle. In this case the complete pipeline is stalled until the delayed read operation is completed successfully.

`if_id_npc (32 bit, in)`

Following address (PC+4) in the decode stage provided by the IF-stage. This address is used by the ID-stage to calculate the destination address in case of a program jump or accepted program branch. See also signal `id_npc` and signal `id_cond`.

`if_id_ir (32 bit, in)`

Current instruction word in the decode stage provided by the IF-stage. The three static instructions formats I, R, J apply (comp. Instruction Set Architecture)

`id_a (32 bit, in)`

Source operand A from the register with address `id_ir_rs1`

`id_b (32 bit, in)`

Source operand B from the register with address `id_ir_rs2`

id_opcode_class (3 bit, out)

Identification of the syntactical class of the current instruction; this signal is used to resolve conflicts in the pipeline issued by data dependencies (Forwarding). All instructions are categorized in one of the following classes:

`NOFORW`	all instructions which don't need a source register (i.e. NOP, J, JAL, ...)
`LOAD`	load instructions (i.e. LW, LH, ...)
`STORE`	store instructions (i.e. SW, SH, ...)
`RR_ALU`	arithmetic/logic instructions with two source registers
`IM_ALU`	arithmetic/logic instructions with one source register and one direct operand
`BRANCH`	jump instructions with source register (i.e. JR, JAR, BEQZ, ...)
`MOVEI2S`	only MOVEI2S
`MOVES2I`	only MOVES2I

`id_ir_rs1 (5 bit, out)`

Part of the instruction word vector (Bit 6..10); Contains the address of the first operand register (A) and is therefore needed to address the register memory.

`id_ir_rs2 (5 bit, out)`

Part of the instruction word vector (Bit 11..15); Contains the address of the second operand register (B) and is therefore needed to address the register memory.

`id_a_fwd_sel (in)`

Control signal which is set by the pipeline control unit and selects the right offset to calculate the jump destination address.

`ex_mem_alu_out (32 Bit, in)`

Result of the ALU-Calculation of the previous clock cycle.

`id_npc (32 Bit, out)`

In case of an accepted program jump (branch, jump) the next instruction is fetched from the address given by this signal provided by the decode stage. See also signal `id_cond.`

`id_cond (out)`

Control signal which decides whether the following instruction (PC+4) or the instruction on the address in `id_npc` is to be fetched by the IF-stage. The signal therefore represents the boolean result of the jump condition.

`id_illegal_instr (out)`

Indicates the environment a not defined instruction code.

`id_halt (out)`

Indicates the environment (Test bench) by a logical 1 that the program execution was stopped by an Exception. At the moment this is only caused by a *TRAP* command, because there is no exception handling intended yet. At the moment this command is used only to stop the simulation (DLX *TRAP* instruction is last command in a machine program)

`id_ex_a (32 Bit, out)`

Pipeline-Reg. of the source operand A from the register with the address `id_ir_rs1`

`id_ex_b (32 Bit, out)`

Pipeline-Reg. of the source operand B from the register with the address `id_ir_rs2`

`id_ex_imm (17 Bit, out)`

Pipeline-Reg. of the direct operand as component of the command word, sign extended.

`id_ex_alu_func (4 Bit, out)`

Pipeline-Reg. of the signal bundle for triggering the ALU in the execution phase.

`id_ex_alu_opb_sel (out)`

Pipeline-Reg. of the control signal to select operand B in the EX-stage

`id_ex_dm_en (out)`

Pipeline-Reg. for the enable signal of the data memory; this enable signal must be set logic 1 for read and write operations.

`id_ex_dm_wen (out)`

Pipeline-Reg. for the write enable of the data memory. See also signal `id_ex_dm_en`.

`id_ex_dm_width (out)`

Pipeline-Reg. which determines, whether the data memory is accessed in Byte-, halfword- or word-format

`id_ex_us_sel (out)`

Pipeline-Reg. which distinguishes between unsigned and signed arithmetic in the EX-stage.

`id_ex_data_sel (out)`

Pipeline-Reg. for source selection for the register (rd) to store the date during the MEM-phase

`id_ex_reg_rd (5 Bit, out)`

Pipeline-Reg. for the destination register address for the date which is written during the MEM phase

`id_ex_reg_wen (out)`

Pipeline-Reg. for the write enable signal of the register memory

`id_ex_opcode_class (out)`

Pipeline-Reg., comp. `id_opcode_class`

`id_ex_ir_rs1 (5 Bit, out)`

Pipeline-Reg., comp. `id_ir_rs1`

`id_ex_ir_rs2 (5 Bit, out)`

Pipeline-Reg., comp. `id_ir_rs2`

Appendix – Sample Program for testing (prime number calculation)

```
main:       addui r1, r0, 1      ; r1: Konstante 1            00000000 24010001
            lw    r2, N(r0)      ; r2: N                      00000004 8c02009c
            addui r3, r0, A      ; r3: Adresse von A[0]       00000008 24030400
            addui r5, r0, (A+2)  ; r5: Basisadresse von A[2]  0000000c 24050402

; Initialisierungsschleife
; ========================
;  for( a[1]=0, i=2; i<= N; i++ )
;    a[i] = 1;
            addui r4, r0, 3      ; r4: i = 3                  00000010 24040003
            sb    1(r3), r0      ; A[1] = 0                   00000014 a0600001
init_loop:  sb    0(r5), r1      ; A[i] = 1                   00000018 a0a10000
            sle   r6, r4, r2     ; r6: i <= N ?               0000001c 0082302c
            addu  r5, r3, r4     ; r5 : Adresse von A[i]      00000020 00642821
            addui r4, r4, 1      ; i++                        00000024 24840001
            bnez  r6, init_loop  ; if r6 == 1  loop init_loop 00000028 14c0ffec
            nop                  ;                            0000002c 0000003f

; Hauptschleife
; =============
; for( i=2; i<=N/2; i++ )
;   for( j=i+i; j<=N; j+=i )
;     a[j] = 0;
            addui r7, r0, (A+4)  ; r7: Basisadresse von A[4]  00000030 24070404
            addui r6, r0, 4      ; r6: j = 4 (= 2*i)          00000034 24060004
            addui r4, r0, 2      ; r4: i = 2                  00000038 24040002
            srai  r5, r2, 1      ; r5: N/2                    0000003c 5c450001
inner:      sb    0(r7), r0      ; A[j] = 0                   00000040 a0e00000
            sle   r8, r6, r2     ; r8: j <= N ?               00000044 00c2402c
            addu  r7, r3, r6     ; r7 : Adresse von A[j]      00000048 00663821
            addu  r6, r6, r4     ; j = j + i                  0000004c 00c43021
            bnez  r8, inner      ; if r8 == 1 loop inner      00000050 1500ffec
            nop                  ;                            00000054 0000003f
            addui r4, r4, 1      ; i++                        00000058 24840001
            nop                  ;                            0000005c 0000003f
            nop                  ;                            00000060 0000003f
            sle   r8, r4, r5     ; r6: i <= N ?               00000064 0085402c
            slli  r6, r4, 1      ; r6; j = 2*i;               00000068 50860001
            nop                  ;                            0000006c 0000003f
            nop                  ;                            00000070 0000003f
            addu  r7, r3, r6     ; r7 : Adresse von A[j]      00000074 00663821
            nop                  ;                            00000078 0000003f
            bnez  r8, inner      ; if r8 == 1  loop inner     0000007c 1500ffc0
            nop                  ;                            00000080 0000003f
            nop                  ;                            00000084 0000003f
            nop                  ;                            00000088 0000003f
            nop                  ;                            0000008c 0000003f
            nop                  ;                            00000090 0000003f
            nop                  ;                            00000094 0000003f
            trap  0              ;                            00000098 44000000
N:          .word   100          ;                            0000009c 00000064
            .align  10
A:          .space  101
```

Literature

[HEN/PAT96]	John L. Hennessy, David A. Patterson Computer Architecture - A Quantitative Approach, 2. Edition Morgan Kaufmann Publishers, Inc. San Francisco, California
[SAI/KAE]	Philip M. Sailer, David R. Kaeli The DLX Instruction Set Architecture Handbook Morgan Kaufmann Publishers, Inc., San Francisco, California