EDA322Digital DesignLab

ChAccProcessor

Designed by: Angelos Arelakis; Anurag Negi, Ioannis Sourdis

This document provides the specifications of the ChAcc (Chalmers Accumulator) processor that will be implemented, evaluated in terms of performance, area and power dissipation and finally downloaded on an FPGA during the 7 lab sessions of the course Digital Design(EDA322).

The ChAcc processor, which is based on the lab processor of HY-120 course in the institute of Computer Science in FORTH, Greece, is a simple but slow processor and can run a variety of programs. It is an 8-bit processor meaning that it executes operations on 8-bit data but using12-bit wide instructions. ChAcc makes use of the accumulator architecture, which has only one special register that is called Accumulator. The Accumulator keeps the result of the most recent operation, while almost every operation works on the Accumulator and the content of a memory location. The Accumulator is so named because in this kind of architectures, it is possible to perform consecutive operations (e.g., additions) and accumulate the result to this register.

This document describes the ChAcc processor and provides important details regarding the Instruction Set Architecture (ISA) and the control signals. This document is organized in the following manner. It first presents the processor's datapath, where it briefly describes the contained components. Then it continues with the presentation of the ISA, where it discusses the syntax and the use of the instructions.At the end, the use of the controller is described detailing the set of control signals and when they must be set/reset so that ChAcc can correctly function.

Section 1 – Datapath

The ChAcc datapath is depicted in Figure 1. The datapath consists of many different units such as an adder, an Arithmetic and Logic Unit (ALU), a bus, memories, muxes, registers and 7-segment displays, while on the upper part of the figure we find the controller. The controller is the brain of the processor since it orchestrates the different units based on the executed instruction (see ISA section).Please refer to the last section for more details about the controller.

The ChAcc processor, like any processor, runs a set of instructions or in other words a program. The program is executed instruction-by-instruction. An instruction execution can be summarized in the following steps:

1. The instruction is read from the instruction memory using the current program counter (PC) as index address.
2. The instruction is split into opcode and instruction arguments and decoded by the controller to figure out which processor's units will be used and which control signals must be set during the whole instruction's execution.
3. Data are fetched using the address part of the instruction from the data memory (except for the jump/branch instructions).
4. Finally, the instruction is executed using the Arithmetic-Logic Unit (ALU). The result is saved into the Accumulator and may be also displayed onto a display.

The rest of this section focuses on particular components of the datapath.

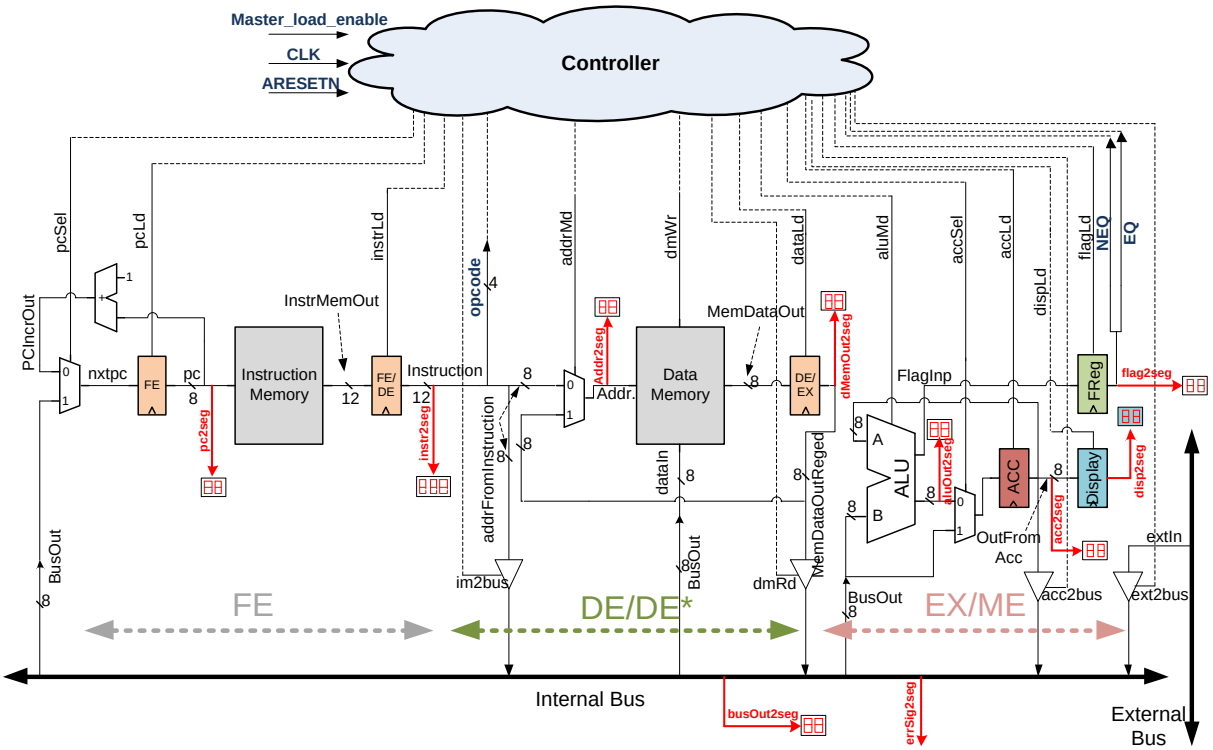

Figure 1: ChAcc processor datapath

Memory

The program's instructions and data are saved into the memory. ChAcc follows the Harvard architecture where the memory is organized in two separate memories: an instruction memory (IM) and a data memory (DM). Both instruction and data memories are accessed using an 8-bit

address (implying both memories have 256 entries) and can be initialized using an initialization file (memory initialization file – mif).

The instruction memory stores the program instructions containing opcode and data memory address (12 bits) while the data memory stores the data (8 bits). Hence the instruction memory has a size of 384B while the data memory has a size of 256B. Finally, the instruction memory is a Read Only Memory (ROM), meaning that cannot be written at run-time, while the data memory can be both read and written at run-time. The memory write is synchronous. On the other hand, the read is implemented in an asynchronous way, but the memory read's output is eventually connected to a register. For example, observe that the "MemDataOut" and "InstrMemOut", which are the outputs of the Data Memory and Instruction Memory, respectively, are connected to the registers "FE/DE" and "DE/EX", respectively, as depicted in Figure 1.

Registers

The registers of the ChAcc processor are the following:

1. Accumulator (ACC): It is the main register of the ChAcc processor.
2. "FE", "FE/DE", "DE/EX":The datapath is divided in several stages[1], thus registers are needed to separate the various stages. "FE/DE" and "DE/EX" registers are named based on which datapath stages they separate. For example, the register between the Fetch and Decode stage is named "FE/DE". On the other hand, "FE" is placed before the Fetch stage.
3. Flag Register (FReg): It keeps the following4 flags inmost significant bit(MSB) toleast significant bit(LSB) order:
    a. *Ovf*[2]: It indicates overflow in the ALU operation.
    b. *NEQ*[3]: Indicates that the two ALU input operands are not equal.
    c. *EQ*3: Indicates that the two ALU input operands are equal.
    d. *Zero*2: Indicates that the ALU data output is zero.
4. Display: The Display register saves the content of the Accumulator if we decide to show its value onthe FPGA's display, using the respective instruction DS.

Looking at Figure 1, we notice that all registers take an input signal from the controller. Based on this signal, the register either maintains the current value, or it updates it with a new one. Thus, the register is implemented as a mux connected to a flip-flop, as is depicted in Figure 2.

[1]An instruction execution is divided in phases. Each phase is executed in one processor's stage (one clock cycle).
[2]"Ovf" and"Zero" are connected to two FPGA leds (Not shown in Figure 1).
[3]"NEQ" and"EQ" are used by the controller in certain situations. Please refer to Section "Controller" for more details.

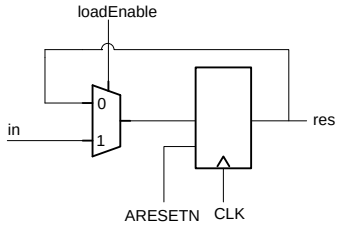

Figure 2: Register –D flip-flop with load enable

Arithmetic and Logic Unit (ALU)

The datapath contains an Arithmetic and Logic Unit (ALU) that can perform all the necessary arithmetic and logic operations. In most modern processors, the ALU can perform addition, multiplication, division between integer and floating point operands, and all logic operations. However, the ALU of theChAcc processor is rather simple and only performs addition as well as few logic operations (and, not, and compare), as depicted in Figure 3.

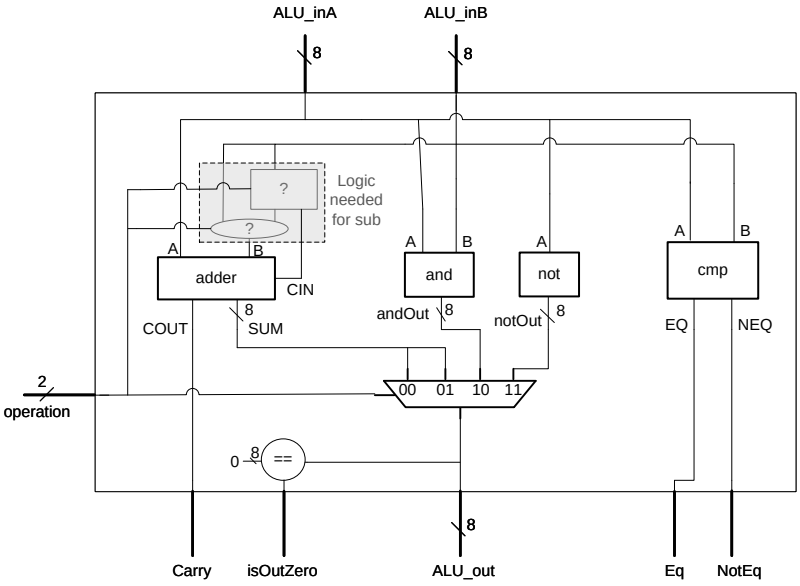

Figure 3: Block diagram of the ALU

The ALU has three inputs and two outputs:

1. Two **inputs** named as"ALU_inA" and "ALU_inB",for the data operands (8 bits).
2. One **input** named as"operation",for the control signal(2 bits) that determines the ALU operation.
3. One **output** (8 bits) named as "ALU_out" and connected to the ACC register,for the result of the operation.
4. Four 1-bit **outputs** that are connected to the respective flags of the Flag Register (FReg) in the following **MSB to LSB** bit order:
    - *Carry*: The carry-out (COUT) of the adder.
    - *NotEq*: Output of the comparator unit (cmp). Set when the input data operands are not equal.
    - *Eq*: Output of the comparator unit. Set when the input data operands are equal.

- *isOutZero*: Set when the ALU data output is zero.

Bus

On the bottom of Figure 1, we can see the bus implemented with tri-states buffers.Each tri-state buffer is driven by a control signal. However,this bus implementation with tri-state buffers is not preferred because if more than one tri-state buffers are on, the bus will take an undefined value. In the lab3 assignment, an alternative bus implementation is presented using multiplexors instead.

The bus has 8 inputs (4 data inputs and 4 control inputs) and 2 outputs:

1. *addrFromInstruction*: The source of this **data input** is the 8 LSB of the instruction as it is output by the "FE/DE" register.
2. *MemDataOutReged*: The source of this **data input** is the output of "DE/EX".
3. *OutFromAcc*: The source of this **data input** is the output of the ACC register.
4. *extIn*: The source of this **data input** is the output of the external bus.
5. *im2bus*: **Control input**. When enabled the bus output is *addrFromInstruction*.
6. *dmRd*: **Control input**. When enabled the bus output is *MemDataOutReged*.
7. *acc2bus*: **Control input**. When enabled the bus output is *OutFromAcc*.
8. *ext2bus*: **Control input**. When enabled the bus output is *extIn*.
9. *busOut2seg*: The bus data **output**.
10. *errSig2seg*: The error **output** of the bus. It is set when two or more control inputs of the bus are set.

7-segment displays

Many datapath signals are connected to 7-segment displays, as depicted in Figure 1. These displays can be used by the user to track the value of particular signals or registers, when the processor is running, to verify the correct operation. The 7-segment displays are very useful when debugging the design.

Top-level design

The entity of the top-level design of the ChAcc processor's datapath is presented in Figure 4. In this code snippet, the inputs/outputs as well as their data width are provided. The signals that are driven to the 7-segment displays have self-explanatory names. All the synchronous circuits are clocked (on the rising edge) with the signal CLK. The reset(ARESETN) is asynchronous and negatively set (set when '0').The use of *master_load_enable* is described in Section 3.

```
entity EDA322_processor is
    Port (   externalIn : in  STD_LOGIC_VECTOR (7 downto 0); -- "extIn" in Figure 1
             CLK : in STD_LOGIC;
             master_load_enable: in STD_LOGIC;
             ARESETN : in STD_LOGIC;
             pc2seg : out  STD_LOGIC_VECTOR (7 downto 0); -- PC
             instr2seg : out  STD_LOGIC_VECTOR (11 downto 0); -- Instruction register
             Addr2seg : out  STD_LOGIC_VECTOR (7 downto 0); -- Address register
             dMemOut2seg : out  STD_LOGIC_VECTOR (7 downto 0); -- Data memory output
             aluOut2seg : out  STD_LOGIC_VECTOR (7 downto 0); -- ALU output
             acc2seg : out  STD_LOGIC_VECTOR (7 downto 0); -- Accumulator
             flag2seg : out  STD_LOGIC_VECTOR (3 downto 0); -- Flags
             busOut2seg : out  STD_LOGIC_VECTOR (7 downto 0); -- Value on the bus
             disp2seg: out STD_LOGIC_VECTOR(7 downto 0); --Display register
             errSig2seg : out STD_LOGIC; -- Bus Error signal
             ovf : out STD_LOGIC; -- Overflow
             zero : out STD_LOGIC); -- Zero
end EDA322_processor;
```

Figure 4: The "entity" VHDL code of the Top-level design

Section 2 – Instruction Set Architecture (ISA)

The ChAcc processor uses its own Instruction Set Architecture (ISA). The ISA is the set of instructions that this processor can recognize and execute.

The ISA of ChAcc is shown in Table 1. The table contains the following columns:

1. Machine code: The binary code of an instruction.
2. Instruction name: The name of the instruction.
3. The instruction written in assembly language format.
4. A brief description of the instruction.
5. Some extra information that must be taken into consideration in particular cases.

Table 1: ISA of the Accumulator architecture

Machine code	Instruction Name	Assembly language	Comment	Extra info
000000000000	No operation	NOOP	Do nothing	–
0001aaaaaaaa	Add	AD ACC, DM[Addr]	ACC = ACC + DataMem[Addr]	may set "Ovf", "Zero"
0010aaaaaaaa	Subtract	SU ACC, DM[Addr]	ACC = ACC - DataMem[Addr]	may set "Ovf", "Zero"
0011aaaaaaaa	AND	AND ACC, DM[Addr]	ACC = ACC &DataMem[Addr]	mayset"Zero"
010000000000	NOT	NT ACC	ACC = ACC'	may set "Zero"
0101aaaaaaaa	Compare	CMP ACC, DM[Addr]	Compare ACC vs. DM[Addr]	set EQ, NEQ
0110aaaaaaaa	Load Byte	LB ACC, DM[Addr]	Load 8 byte value from location DataMem[Addr] into ACC	–
0111aaaaaaaa	Store Byte	SB DM[Addr], ACC	Store contents of ACC into location DataMem[Addr]	–
1000aaaaaaaa	Add Index	ADX ACC, DM[DM[Addr]]	ACC = ACC + DataMem[DataMem[Addr]]	may set "Ovf", "Zero"
1001aaaaaaaa	Load Byte Index	LBX ACC, DM[DM[Addr]]	ACC = DataMem[DataMem[Addr]]	–
1010aaaaaaaa	Store Byte Index	SBX DM[DM[Addr]], ACC	DataMem[DataMem[Addr]] = ACC	–
1011aaaaaaaa	Input	IN DM[Addr], IO_BUS	DataMem[Addr] = value at IO_BUS	–
1100aaaaaaaa	Jump	J Addr	Execute next instruction @ PC = Addr	–
1101aaaaaaaa	Jump Not Equal	JNE Addr	Jump if the corresponding flag NEQ is set	–
1110aaaaaaaa	JumpEqual	JEQ Addr	Jump if the corresponding flag EQ is set	–
111100000000	Display	DS	Move ACC to Display reg. (used for debugging)	–

ChAcc makes use of an ISA with only 16 instructions. According to the first column of Table 1 that shows the machine code for all the instructions, each instruction is 12 bits wide. The 4 MSB of the instruction compose the opcode (operation code), while the 8 LSB ("aaaaaaaa" at Table 1) form the **address** that is used to access the data memory. The opcode is each instruction's unique code, while the use of the address is explained later. Instructions like *NOT, NOOP* and *DS* have a zero address field, as they don't need to access the data memory.

The ISA (of Table 1) primarily consists of three groups of instructions:

1) **Arithmetic and logic instructions**: The instructions *Add*, *Subtract*, *AND*, *NOT*, *Compare* and *Add Index* belong to this group. These instructions make use of the ALU unit and perform arithmetic or logic operations between the ACC and the content of a data memory location (except the *NOT* instruction). The address-field of the instruction is used to access the data memory and retrieve the second operand of the ALU.
2) **Memory instructions**: The instructions *Load Byte*, *Store Byte*, *Load Byte Index*, *Store Byte Index* and *Input* that belong to this group access the data memory using the address-field of the instruction as an index. Memory instructions can:
   a) read something from the data memory and save it to the ACC (*Load Byte*, *Load Byte Index*),
   b) write the content of the ACC into the data memory (*Store Byte*, *Store Byte Index*),or
   c) write the data that come from the I/O bus into the data memory.
3) **Jump instructions**: The instructions *Jump*, *Jump Equal* and *Jump Not Equal* that belong to this group can change the program flow by modifying the program counter (PC) based on a condition (*JE*, *JNE*) or unconditionally (*J*), by jumping to a particular address. The address-field of the instruction is used to change the program flow.

Moreover, there are other instructions that do not belong to any of the groups above. The *Display* (*DS*) instruction is used for debugging by moving the content of the ACC register into the Display register (shaded with blue color in Figure 1), while the *NOOP* is used to keep the processor idle. However, the *NOOP* operation is implemented like an *add* instruction between the ACC and the first location of the data memory (DM[0]), which is assumed that is always 0 and *should not be modified*. Thus, the NOOP doesn't do any useful operation since it adds zero to the current content of the ACC. Finally, note that the instructions *ADX*, *LBX* and *SBX*access the data memory *twice*.

Section 3 – Controller

In any processor, a special unit is needed in order to synchronize the rest of the units and orchestrate their operations. This unit is called controller and is actually the "brain" of a processor. In the ChAcc processor, the controller is shown on the top of Figure 1.

Controller's Interface

As is shown in this figure, the controller has the following **inputs**:

1. *opcode*: The 4 MSB of the currently decoded instruction are used by the controller in order to determine the currently to-be-executed instruction and set/reset the particular signals and enable/disable particular parts of the datapath during the phases of the instruction.
2. The signals *NEQ* and *EQ* output from the "FReg".

3. *CLK*: The processor's clock.
4. *ARESETN*: The processor's reset signal.
5. *master_load_enable*: This signal is connected to an FPGA switch that is set by the user and plays the role of a **manual** clock toggling. In other words, by toggling this signal, the user is able to control the clocking of the design, "freezing" and "starting" the time. This is useful when debugging the design; otherwise the changes on the displays would not be visible to a human's eye, as the design's clock is on the order of hundreds of MHz. The *master_load_enable* affects the following:
    a. The internal state transitions of the controller (the controller is implemented as a Finite State Machine (FSM), as is described in lab4 assignment) are enabled when *master_load_enable* is set.
1. The registers save their input onthe rising clock edge when *master_load_enable* is set and if the respective control signal of a register is also set. Thus the*master_load_enable* must be combined with the respective control signal (see *pcSel* and *pcLd*: Control the logic that is relevant to the program counter (PC).
2. *instrLd*: Is the "load enable" signal of the register that keeps the instruction read. It's the only signal that is always set during the whole execution of all instructions. This happens because according toTable 2, all instructions go through the Fetch (FE) and Decode (DE) stages. During the FE stage, the same action is taken for all the instructions: the instruction must be read from the instruction memory using the PC as index, and must be saved in the "FE/DE" register. Thus the *instrLd* signal, which enables the "FE/DE", is set during this stage for every instruction and can remain set for all the stages, as the index of the Instruction Memory (saved in "FE") remains the same too.
3. *addrMd*: Controls the data memory's index thatcan be input by two different sources. If the instruction is a non-index instruction, the data memory is accessed using the instruction's address field. If the instruction is an index instruction, the data memory is normally accessed using instruction's address in DE stage. However, when the second memory access takes place during the *DE** stage, the source of the data memory index is instead the output of the data memory itself.
4. *dmWr*: Enables the write function of the data memory, when set.
5. *dmRd*: Enables the read function of the data memory, when set.
6. *dataLd*: Is the "load enable" signal of "DE/EX" that saves the read memory value.
7. *flagLd*: Is the "load enable" signal of "FReg" that saves the *Flags*.
8. *accSel* and *accLd:* Control the logic of the ACC register. The *accLd* is the "load enable" signal of ACC, while the *accSel* controls the source of its input. ACC is loaded with data that produced either from the ALU or come from the bus if it is a store instruction.
9. *dispLd*: Enables the load of the display register. The display register is used for buffering values for display on a7-segment display available on the FPGA board.
10. *im2bus*, *dmRd*, *acc2bus* and *ext2bus*: Are the control signals of the bus.
11. *aluMd*: Determines the ALU operation.

b. Table 3) to eventually drive each register's *load enable* (see Figure 2).

The controller **outputs** the rest of the depicted signals (in Figure 1), which control the muxes, the memories, the bus, the registers, the ALU and in general all the datapath modules.

Stages

As it is shown on the bottom of Figure 1, the datapath of the ChAcc processor is divided into five stages by the controller, as the various instructions make use of different datapath modules in order to be executed. Thus the datapath is divided into many stages so that every instruction is executed by utilizing only the useful stages. The 5 stages of the datapath are:

1. Fetch (FE): The instruction is fetched from the instruction memory using the program counter (PC) as an address.
2. Decode (DE): The instruction is decoded and the data memory is read.
3. Decode* (DE*): The data memory is read for a second time.
4. Execute (EX):The ALU operation takes place and the result is written to ACC.
5. Memory (ME): A previously calculated result (already saved in ACC) is written back to the data memory.

Every stage has a duration of one clock cycle. The clock cycle time is determined by the latency of the slowest datapath stage (critical path). If the whole datapath was clocked as one large stage, then all the instructions would have the same execution time resulting in a simpler controller design. However, it is more advantageous to have a multi-stage datapath as different instructions of the ISA utilize a variable number of datapath stages, thus require a variable number of clock cycles, resulting in different execution time among them. This can potentially yield a more efficient design in terms of performance. Finally, a multi-stage datapath can be more easily pipelined to parallelize the execution of more instructions per cycle. However, the latter requires computer organization knowledge and is out of the scope of this course.

Table 2 summarizes the stages utilized by the different instructions marking with 'y' (yes) the used ones and with 'n' (no) the unused stages. The DE* stage is required only by the index instructions (*ADX*, *LBX*, *SBX*) because these instructions access the data memory twice but this cannot happen in the same cycle. The last column of the table presents the actual number of used stages (cycles needed) per instruction. Looking at Table 2, we can conclude that although the total number of datapath stages is 5, the longest executed instructions that are the Index instructions make use of 4 datapath stages, while there are instructions that need only two cycles to be executed.

Table 2:Datapath stages per instruction

opcode	detailedinstr	FE	DE	DE*	EX	ME	#stages
0000	AD ACC, DM[0]	y	y	n	y	n	3
0001	AD ACC, DM[Addr]	y	y	n	y	n	3
0010	SU ACC, DM[Addr]	y	y	n	y	n	3
0011	AND ACC, DM[Addr]	y	y	n	y	n	3
0100	NOT ACC	y	y	n	y	n	3
0101	CMP ACC, DM[Addr]	y	y	n	y	n	3
0110	LB ACC, DM[Addr]	y	y	n	y	n	3
0111	SB DM[Addr], ACC	y	y	n	n	y	3
1000	ADX ACC, DM[DM[Addr]]	y	y	y	y	n	4
1001	LBX ACC, DM[DM[Addr]]	y	y	y	y	n	4
1010	SBX DM[DM[Addr]], ACC	y	y	y	n	y	4
1011	IN DM[Addr], IO_BUS	y	y	n	n	n	2
1100	J Addr	y	y	n	n	n	2
1101	JNE Addr, NEQ	y	y	n	n	n	2
1110	JEQ Addr, EQ	y	y	n	n	n	2
1111	DS	y	y	n	y	n	3

Control signals

As different instructions make use of different datapath stages, the controller must determine which datapath stage is used by an instruction and when (which cycle), by setting/resetting particular signals that control the various datapath modules.

12. *pcSel* and *pcLd*: Control the logic that is relevant to the program counter (PC).
13. *instrLd*: Is the "load enable" signal of the register that keeps the instruction read. It's the only signal that is always set during the whole execution of all instructions. This happens because according toTable 2, all instructions go through the Fetch (FE) and Decode (DE) stages. During the FE stage, the same action is taken for all the instructions: the instruction must be read from the instruction memory using the PC as index, and must be saved in the "FE/DE" register. Thus the *instrLd* signal, which enables the "FE/DE", is set during this stage for every instruction and can remain set for all the stages, as the index of the Instruction Memory (saved in "FE") remains the same too.
14. *addrMd*: Controls the data memory's index thatcan be input by two different sources. If the instruction is a non-index instruction, the data memory is accessed using the

instruction's address field. If the instruction is an index instruction, the data memory is normally accessed using instruction's address in DE stage. However, when the second memory access takes place during the *DE** stage, the source of the data memory index is instead the output of the data memory itself.

15. *dmWr*: Enables the write function of the data memory, when set.
16. *dmRd*: Enables the read function of the data memory, when set.
17. *dataLd*: Is the "load enable" signal of "DE/EX" that saves the read memory value.
18. *flagLd*: Is the "load enable" signal of "FReg" that saves the *Flags*.
19. *accSel* and *accLd:* Control the logic of the ACC register. The *accLd* is the "load enable" signal of ACC, while the *accSel* controls the source of its input. ACC is loaded with data that produced either from the ALU or come from the bus if it is a store instruction.
20. *dispLd*: Enables the load of the display register. The display register is used for buffering values for display on a7-segment display available on the FPGA board.
21. *im2bus*, *dmRd*, *acc2bus* and *ext2bus*: Are the control signals of the bus.
22. *aluMd*: Determines the ALU operation.

Table 3 depicts the values of the control signals for every instruction. The first column of the table presents the "opcode" (of the decoded instruction) while the rows summarizes all the control signals for the opcodes. The notation used follows the "X_Y" format, where X is the signal's value and Y is the stage which the signal must take this value at. For example, the signal *flagLd* is set at stage EX (Execute) when the opcode of the instruction is "0000", otherwise it is 0.However, in cases when a signal is set or unset during the whole execution of an instruction, the value X is only presented, e.g., *acc2bus*is '1' during the whole execution of instruction with opcode "1010". Note also that the value of a signal may be 'x' instead of '1' or '0'. This means that the signal can take any value (*don't care*). The control signals are the following:

23. *pcSel* and *pcLd*: Control the logic that is relevant to the program counter (PC).
24. *instrLd*: Is the "load enable" signal of the register that keeps the instruction read. It's the only signal that is always set during the whole execution of all instructions. This happens because according toTable 2, all instructions go through the Fetch (FE) and Decode (DE) stages. During the FE stage, the same action is taken for all the instructions: the instruction must be read from the instruction memory using the PC as index, and must be saved in the "FE/DE" register. Thus the *instrLd* signal, which enables the "FE/DE", is set during this stage for every instruction and can remain set for all the stages, as the index of the Instruction Memory (saved in "FE") remains the same too.
25. *addrMd*: Controls the data memory's index thatcan be input by two different sources. If the instruction is a non-index instruction, the data memory is accessed using the instruction's address field. If the instruction is an index instruction, the data memory is normally accessed using instruction's address in DE stage. However, when the second memory access takes place during the *DE** stage, the source of the data memory index is instead the output of the data memory itself.

26. *dmWr*: Enables the write function of the data memory, when set.
27. *dmRd*: Enables the read function of the data memory, when set.
28. *dataLd*: Is the "load enable" signal of "DE/EX" that saves the read memory value.
29. *flagLd*: Is the "load enable" signal of "FReg" that saves the *Flags*.
30. *accSel* and *accLd:* Control the logic of the ACC register. The *accLd* is the "load enable" signal of ACC, while the *accSel* controls the source of its input. ACC is loaded with data that produced either from the ALU or come from the bus if it is a store instruction.
31. *dispLd*: Enables the load of the display register. The display register is used for buffering values for display on a7-segment display available on the FPGA board.
32. *im2bus*, *dmRd*, *acc2bus* and *ext2bus*: Are the control signals of the bus.
33. *aluMd*: Determines the ALU operation.

Table 3: Control signals per instruction

opcode	pcSel	pcLd	instrLd	addrMd	dmWr	dataLd	flagLd	accSel	accLd	im2bus	dmRd	acc2bus	ext2bus	dispLd	aluMd
0000	0	1_EX	1	0	0	1_DE	1_EX	0	1_EX	0	1_EX	0	0	0	00
0001	0	1_EX	1	0	0	1_DE	1_EX	0	1_EX	0	1_EX	0	0	0	00
0010	0	1_ EX	1	0	0	1_DE	1_EX	0	1_EX	0	1_EX	0	0	0	01
0011	0	1_ EX	1	0	0	1_DE	1_EX	0	1_EX	0	X	0	0	0	10
0100	0	1_ EX	1	0	0	1_DE	1_EX	0	1_EX	0	1_EX	0	0	0	11
0101	0	1_ EX	1	0	0	1_DE	1_EX	0	0	0	1_EX	0	0	0	xx
0110	0	1_ EX	1	0	0	1_DE	0	1_EX	1_EX	0	1_EX	0	0	0	xx
0111	0	1_ME	1	0	1_ME	0	0	0	0	0	0	1	0	0	xx
1000	0	1_EX	1	1_DE*	0	1_DE/ 1_DE*	1_EX	0	1_EX	0	1_EX	0	0	0	00
1001	0	1_EX	1	1_DE*	0	1_DE/ 1_DE*	0	1_EX	1_EX	0	1_EX	0	0	0	xx
1010	0	1_ME	1	1_ME	1_ME	1_DE	0	0	0	0	0	1	0	0	xx
1011	0	1_DE	1	0	1_DE	0	0	0	0	0	0	0	1	0	xx
1100	1_DE	1_DE	1	0	0	0	0	0	0	1	0	0	0	0	xx
1101	1_DE[4]	1_DE	1	0	0	0	0	0	0	1	0	0	0	0	xx
1110	1_DE[5]	1_DE	1	0	0	0	0	0	0	1	0	0	0	0	xx
1111	0	1_EX	1	0	0	0	0	0	0	0	0	0	0	1_EX	00

[4]If the instruction is JNE, the value of *pcSel* at the DE stage is additionally affected by signal NEQ.

[5]If the instruction is JEQ, the value of *pcSel* at the DE stage is additionally affected by signal EQ.

Let's take some example instructions, to explain how particular control signals are set. You can better comprehend these examples by looking at the datapath animations in the slides of Lecture 4 or the uploaded document "example_commands.pdf".

A first example isan *Add Index* instruction. First of all, according to Table 2, the *ADX*(opcode: 1000) uses 4 stages: FE, DE, DE* and EX. It was previously explained what happens during the FE stage. Looking at the line with opcode "1000" in Table 3 we can see the exact value of all the signals during all the stages. As the *ADX* instruction (like all index instructions) accesses the data memory twice, it needs the output of the data memory as an address in the DE* stage. Thus, during the DE* stage, the *addrMd* is now set so that the mux (before the data memory) multiplexes the input coming from "DE/EX" as an index to the data memory. The *dataLd* has to be set for both DE and DE* stages to save the data read from the data memory in the "DE/EX". Finally, in the Execute stage, the previously set signals are reset and 4 new signals are set:

- *dmRd*: It drives the read data (saved in the "DE/EX" register) to the ALU (ALU_inB).
- *accLd*: It enables ACC to save the ALU output.
- *flagLd*: It enables FReg to save the *Flags*.
- *pcLd*: It enables the "FE" register to save the PC of the next executed instruction.

The signal *aluMd*, which is connected to the *operation* input of the ALU, is also set to "00" which stands for the add operation. The aluMd encoding is explained in detail in the description of the lab2 assignment.

A second example is the *Store Byte* instruction. According to Table 2, the opcode is "0111" and uses 3 stages: FE, DE and ME. Looking at Table 3, none of the signals is set during the DE stage while during the ME stage, the ACC output is written to the data memory. Therefore, the *dmWr* signal is set so that the data memory can write the data into the memory location that is determined by the instruction address-field. The signal*acc2bus* is also set so that the ACC register's output can be driven to the data memory through the bus. Similarly to the ADX instruction above, the *pcLd* is set to save the PC of the next instruction in the FE register.

A last example is the *JEQ* instruction, which has the "1110" opcode and uses only two stages: FE and DE. In the DE stage, according to Table 3, the *pcLd* and *pcSel* are set so that the address-field of the instruction, which is driven through the bus (notice that *im2bus* is also set), can be saved in the FE register. However, when JEQ and JNE instructions are executed, the EQ and NEQ fields of the "Flags" signal must be evaluated as well to determine whether *pcSel* will be set or not. This takes place inside the controller. If the respective flag (in this example the EQ) is not set, then the *pcSel* (*pcSel* controls the multiplexor on the left of the datapath) must not be set.

Finally, it must be mentioned here that the purpose of this document was to describe the processor's datapath and the specifications of the controller. The specifications and the functionality of particular components, such as the adder or the implementation of the

controller using an FSM, as well as the detailed interfaces (inputs/output names and exact widths)are described in detail in the lab assignments.