Designing a Time Predictable Network on Chip Based Multiprocessor for Real Time Applications

Group 6

Sholto Bolton, Jonathan Browne, Blake Williams

Abstract - This report covers the development and design decisions of developing a time predictable network on chip based multiprocessor for real time applications. The design results in the implementation of a hybrid customisable architecture that utilises multiple different cores in a time division multiple access multistage interconnection network to produce a system that meets high performance in metrics such as response times, predictability and energy consumption. The design produces an oscilloscope like application developed for the DE1-SoC board.

I. Introduction

The purpose of this project is to investigate the use of FPGAs to produce a system that surpasses modern day standard available execution platforms. To surpass these standard execution platforms we require a design that can manage higher computational loads. To achieve this we require a platform that can be customised for specific applications by using processing and storage components that are interconnected to be able to transmit data freely between the units. This type of system could be achieved by the utilisation of multiple different types of cores, which together are able to achieve the targeted system requirements. By implementing a multi-core system our team is required to implement a interconnection network for a Network on Chip execution based platform (NoC).

The aim of this project was to investigate and implement signal processing features for an oscilloscope based design to be run on a Hybrid Customisable Architecture (HyCA). The HyCA utilises multiple core connected networks using Time Division Multiple Access Multistage Interconnection Networks (TDMA_MIN). The design develops a single type of general purpose processor called Reactive and Concurrency-Processor(RECOP). The RECOP is used in the design to read the instruction memory and correspondingly generate configuration packets for the 3 Application Specific Processors (ASPs) as well as handling the configuration of output locations via the TDMA_MIN network. These ASP designs will be used to conduct signal processing on an incoming input waveform and outputs displayed via the seven segment display on the DE1-SoC board as well as generate analog waveforms to be outputted by the DAC. The DE1-SoC board will be used as an interface to send input packets via the switches and buttons located on the board to configure the ASP's, ADC and DAC. The application produced is therefore required to take analog input signals sampled and process those samples using signal processing algorithms or sequential applications of those algorithms. After computation the result will be converted into a digital result into digital display and also set to the DAC to produce analog output waveform. To produce the required application we must therefore produce a ReCOP unit, 3 ASPs and 8 ports in the TDMA-MIN NoC, which allows connecting up to 8 cores, ReCOPs and ASPs.

II. ReCOP

The ReCOP processor was chosen to be a three stage pipeline processor. The instructions are read from ROM and executed on the pipeline. The ReCOP is able to perform many operations that were determined to be necessary for the implementation to function. Below is a table of all implemented instructions on our recop control processor:

Instruction	Description	Register Transfer	Opcode
AND Rz Rx #Operand	The contents of Rx and Operand are ANDed and written into Rz	Rz <- Rx AND Operand	00000000
AND Rz Rz Rx	The contents of Rz and Rx are ANDed and written into Rz	Rz <- Rz AND Rx	00000001
OR Rz Rx #Operand	The contents of Rx and Operand are ORed and written into Rz	Rz <- Rx OR Operand	00000010
OR Rz Rz Rx	The contents of Rz and Rx are ORed and written into Rz	Rz <- Rz OR Rx	00000011
ADD Rz Rx #Operand	The contents of Rx and Operand are ADDed and written into Rz	Rz <- Rx + Operand	00000100
ADD Rz Rz Rx	The contents of Rz and Rx are ADDed and written into Rz	Rz <- Rz + Rx	00000101
SUBV Rz Rx #Operand	The Operand is subtracted from the contents of Rx and written into Rz	Rz <- Rx - Operand	00000110
SUB Rz #Operand	The Operand is subtracted from the contents of Rz but the result is not written	Rz - Operand	00000111
LDR Rz #Operand	The Operand is written into register Rz	Rz <- Operand	00001000
LDR Rz Rx	Memory location given by contents of Rx are loaded into Rz	Rz <- MEM[Rx]	00001001
LDR Rz $Operand	Memory location given by operand is loaded into Rz	Rz <- MEM[Operand]	00001010
STR Rz #Operand	Operand is loaded into memory location given by contents of Rz	MEM[Rz] <- Operand	00001011
STR Rz Rx	Contents of Rx are loaded into memory location given by Rz	MEM[Rz] <- Rx	00001100
STR Rx $Operand	Contents of Rx are loaded into memory location given by Operand	MEM[Operand] <- Rx	00001101
JMP #Operand	Jump to location given by Operand	PC <- Operand	00001110
JMP Rx	Jump to location given by contents of Rx	PC <- Rx	00001111
IOWRITE Rz Rx	Contents of Rx are written into IOMemory at location given by Rz	IOMEM[Rz] <- Rx	00010000
IOWRITE Rz #Operand	Operand is written into IOMemory at location given by Rz	IOMEM[Rz] <- Operand	00010001
IOREAD Rz Rx	Contents of IOMem at location Rx are loaded into register Rz	Rz <- IOMEM[Rx]	00010010
Sz #Operand	Jump to location given by Operand if Z = 1	If Z = 1 then PC <- Operand else NEXT	00010011

CLFZ	Clear the Z flag	Z <- 0	00010100
NOOP	No operation		00010101
MAX	Operand is written into Rz if it is greater than Rz	Rz <- MAX{Rz, Operand}	00010110
STRPC Rz	PC value is written to register Rz	Rz <- PC	00010111
LSL Rz Rx #Operand	Contents of Rx are left-shifted by number specified by operand and written to Rz	Rz <- Rx << Operand	00011000
RSL Rz Rx #Operand	Contents of Rx are right-shifted by number specified by operand and written to Rz	Rz <- Rx >> Operand	00011001

Figure 1. ReCOP Instruction set

The instruction format is similar to the original ReCOP instruction set that it is based on but we opted to remove the first two addressing-mode(AM) bits to simplify the control portion of our processor design, simply hardcoding the value to “00”; this results in an 8-bit unique opcode for each instruction. Basic logical and arithmetic functions are all implemented allowing the processor to manipulate input data from data-processing processors. Additionally left and right logical shift operations are implemented to allow for much easier extraction of input packets; in particular when reading the switch or button values these operations make extracting the value of a particular set of switches much easier (ANDing then shifting right by required amount can directly extract value of switches 3..1 for example). The combination of the non-writing sub instruction and sz jump instruction allow for branching to specific instructions, used specifically in our system for detecting button presses.

“00”	OPCODE(6)	Rz(4)	Rx(4)	Operand(16)

Figure 2. ReCOP Instruction Packet

We chose to map IO to memory locations in an IO register files the registers are then directly written to the output lines of the processor depending on input and current instruction execution. This removes the need for signal control and specific data register instructions and we have replaced these with IOWRITE and IOREAD instructions where the respective control outputs are mapped as follows:

IO Register Address	
A0	NULL
A1	send_data(31 downto 16): the upper 16 bits of the current 32-bit configuration packet
A2	send_data(15 downto 0): the lower 16 bits of the current 32-bit configuration packet
A3	send_addr(8-bits): the target address for the current configuration packet
A4	hexOut0(4-bits): value to be displayed on segment hex0
A5	hexOut1(4-bits): value to be displayed on segment hex1
A6	hexOut2(4-bits): value to be displayed on segment hex2
A7	hexOut3(4-bits): value to be displayed on segment hex3

A8	hexOut4(4-bits): value to be displayed on segment hex4
A9	hexOut5(4-bits): value to be displayed on segment hex5
A10	switches(10-bits): most recently read value of the switches
A11	buttons(4-bits): most recently read value of the buttons

Figure 3. I/O register locations

All registers are 16-bit where the lower portion corresponding to the above sizes is used as the output; addr(1) and addr(2) are concatenated and outputted across send_data as a config packet when a button is pressed, at all other times a data packet containing a 24-bit value corresponding the 6 4-bit hex values is sent to the IONode to be displayed. Input format for buttons and switches used to construct these packets is included in the HyCa section of this report.

Some above instructions; data memory store and load instructions in particular, have not been tested as these were not utilised in the control program used in our system implementation. If more complex control was required the system is in place however to allow storage of input data into data memory. The below figure shows our full program utilizing the above instruction set.

```
0.  IOREAD R1 A10       //read switch value into R1
1.  IOREAD R2 A11       //read button value into R2
2.  NOOP

3.  ANDop R3 R1 896     //extract bits 9..7 target address
4.  ANDop R4 R1 112     //extract bits 6..4 forward address
5.  ANDop R5 R1 14      //extract bits 3..1 mode value
6.  ANDop R6 R1 1       //extract bit 0 enable bit

7.  LSL R7 R3 1         //shift target address to 10..8 of top 16 bit 26..24 of 32-bit
8.  RSL R8 R3 7         //shift target address to bits 2..0 for display
9.  RSL R9 R4 4         //shift forward address to bits 2..0 for display
10. RSL R10 R5 1        //shift mode to bits 2..0 for display
11. IOWRITE A9 R8       //disp target address on hex4
12. IOWRITE A8 R9       //disp forward address on hex2
13. IOWRITE A7 R10      //disp mode on hex0

14. SUB R2 7            //check for KEY3 press ASP active low = 0111
15. SZ 21
16. SUB R2 11           //check for KEY2 press ADC active low = 1011
17. SZ 35
18. SUB R2 13           //check for KEY1 press DAC active low = 1101
19. SZ 46
20. JMPop 0

21. LDRop R11 0         //clear reg 11
22. NOOP                //delay to wait for R11 result
23. NOOP
24. OR R11 R11 R4       //initial lower 16-bits missing mode and en
25. ORop R7 R7 36864    //upper 16-bits of full ASP packet first four 1001 = 36864
26. NOOP                //delay to wait for R11 result
27. OR R11 R11 R5       //lower 16-bits of full ASP packet adding mode
28. NOOP                //delay to wait for R11 result
29. NOOP
30. OR R11 R11 R6       //lower 16-bits of full ASP packet adding en
31. IOWRITE A3 R8       //write target address to addr(3)
32. IOWRITE A1 R7       //write upper 16-bits to addr(1)
33. IOWRITE A2 R11      //write lower 16-bits to addr(2)
```

```
34.  JMPop 0

35.  LDRop R11 0        //clear reg 11
36.  NOOP               //delay to wait for R11 result
37.  NOOP
38.  OR R11 R11 R4      //initial lower 16-bits missing en
39.  ORop R7 R7 40960   //upper 16-bits of full ADC packet first four 1010 = 40960
40.  NOOP                    //wait for R11 result
41.  OR R11 R11 R6      //lower 16-bits of full ADC packet adding en
42.  IOWRITE A3 R8      //write target address to addr(3)
43.  IOWRITE A1 R7      //write upper 16-bits to addr(1)
44.  IOWRITE A2 R11     //write lower 16-bits to addr(2)
45.  JMPop 0

46.  ORop R7 R7 45056   //upper 16-bits of full DAC packet first four 1011 = 45056
47.  NOOP                    //delay to wait for R7 result
48.  NOOP
49.  IOWRITE A3 R8      //write target address to addr(3)
50.  IOWRITE A1 R7      //write upper 16-bits to addr(1)
51.  IOWRITE A2 R6      //write lower 16-bits to addr(2) only include en for DAC
52.  JMPop 0
```

Figure 4. Program Instructions

This first section executes continuously unconditionally obtaining the switch and key values and outputs the switch values to the appropriate hex segments and checks for key presses. The second section (lines 21-34) is for sending of an ASP packet and reached by SZ 21 when a KEY3 press is detected, the ADC and DAC are similar for the third and fourth sections where the ADC is reached on a KEY2 press and the DAC on a KEY1 press. Note the instructions are specifically indicated as op here to avoid confusion i.e. JMPop or LDRop.

The Program Instructions are passed through an assembler where the Assembly is converted into binary. The binary instructions are configured into *instructions.mif*, where each instruction is indexed against the correct address to access the instruction. The *RecopInstructionMemory* component reads the *mif* and turns it into ROM which can be used by the ReCOP processor.

III. HyCA

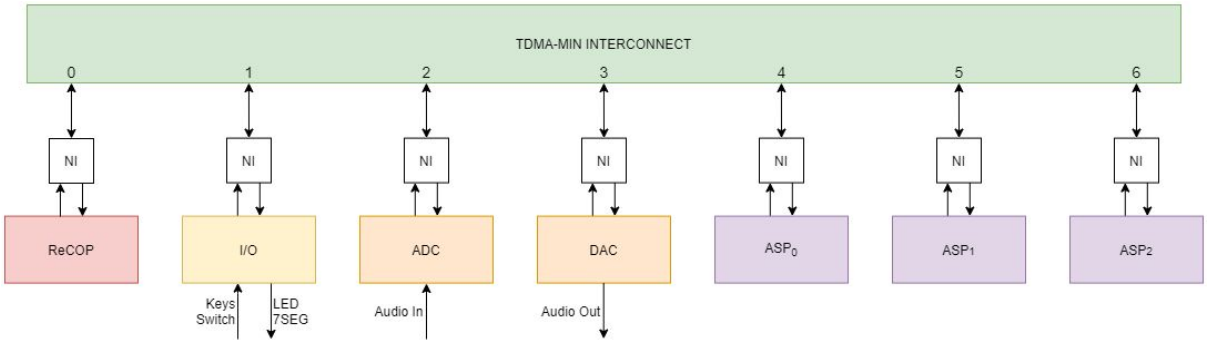

Figure 5. TDMA-MIN Interface

where, ASP_0 = Jbro682_ASP, ASP_1 = Sbol998_ASP, ASP_2 = Bwil410_ASP

The TDMA-MIN has seven nodes connected to it. The ReCOP manages the configuration packets for each of the other six nodes, which allows it to set up the order in which data will flow, such as moving ADC signals through an ASP to the DAC. The I/O node sends user input signals to the ReCOP, which in return sends the required information to the 7-Segment display for the user to read. The ADC and DAC read and write to the audio in/out respectively, and with a given set of configurations will pass and receive data from wherever programmed to.

The configuration packets sent by the ReCOP are crucial for correct operation. Each node has a seperate id-code to indicate what device the instruction format is for.

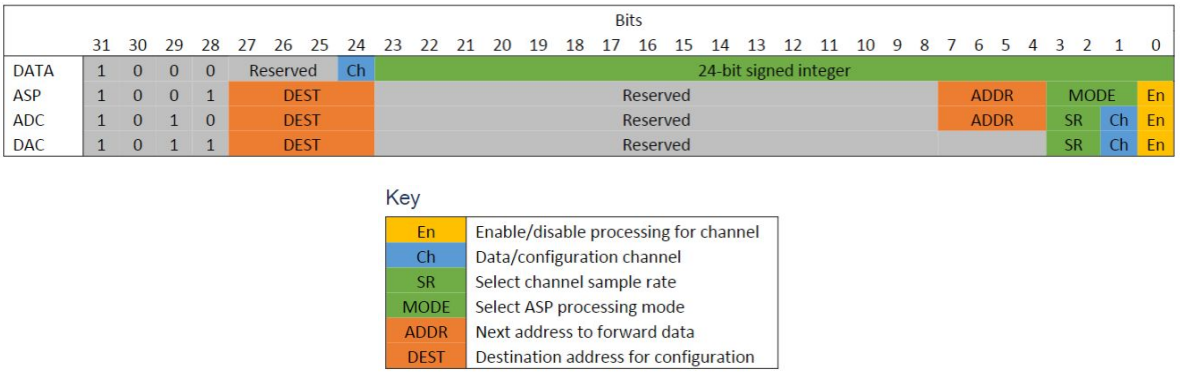

Figure 6. TDMA-MIN Instruction/Data Packets

A. Input-Output Format

The following is the input format of the IO devices on the DE1-soc board that allow the user to send configuration packets to nodes in the HyCa. The seven-segment displays are used to display current and previous configuration information whilst the switches and buttons are used to adjust the input values; the specific format is as follows:

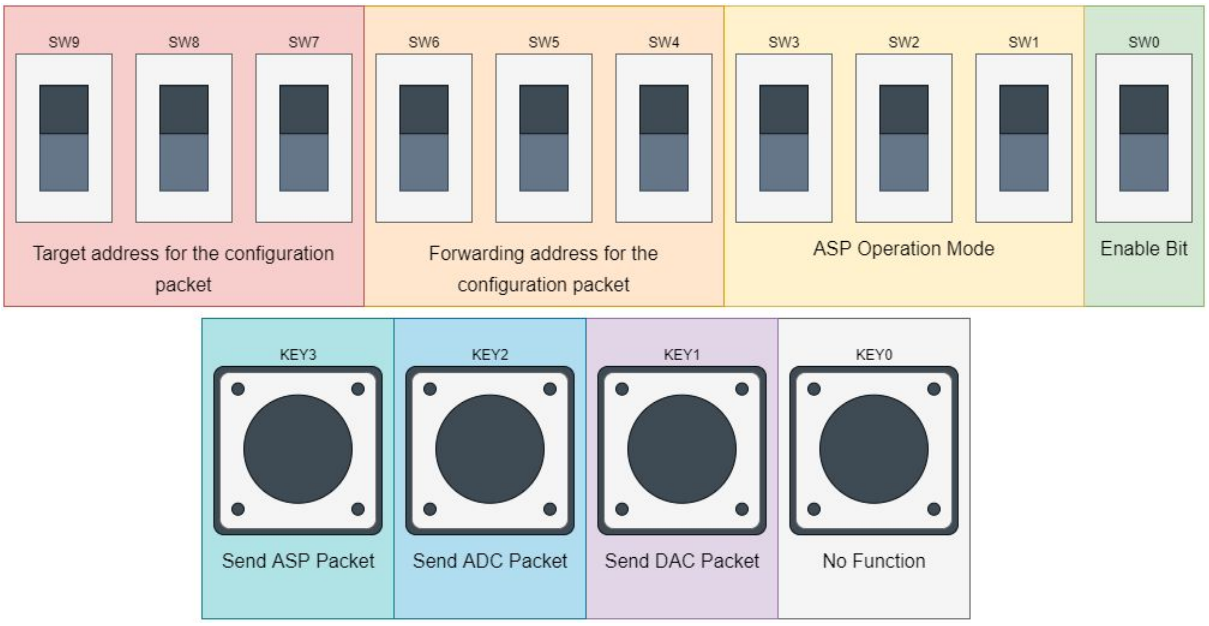

Figure 7. Input configurations from the board

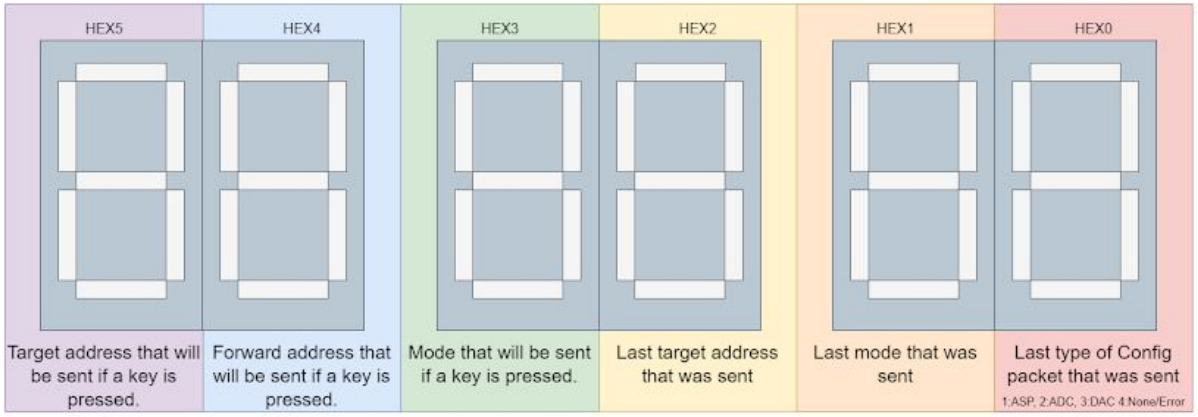

Figure 8. Output display from the HyCA

It is to note that the mode is not used in ADC or DAC packets and Hex1 will be set to 0 when one of these packets is sent regardless of the switch mode settings (same for the forwarding address of DAC although the forwarding address is not displayed). If the target address is set to 0 or 1 the config packet will not be sent and this is reflected by hex2 - hex0 not updating (to prevent unexpected behaviour with the IONode or Recop receiving configuration packets).
The list of mode values for each ASP differ slightly but generally:

- Mode “000” = direct passthrough
- Mode “001” = linear filter
- Mode “010” = maximum peak
- Mode “011” = minimum peak

IV. Demonstrations and evaluations

In terms of completed implementation our final design is able to successfully generate packets from the input switches and keys to configure ASPs, ADC, DAC and ReCOP. We have confirmed this by configuring the ADC data to output to the DAC where the audio output is audible. This implementation proves the correct implementation of the ReCOP protocol and correct implementation of program instructions read by the ReCOP. Additionally, it proves a correct 8 port TDMA-MIN connection as when configured incorrectly no sound is produced on the DAC output. Unfortunately the three ASP designs do not produce audio output when configured with the ADC to ASP to DAC connection. This is likely caused by incorrect implementation of the ASP design and not due to our HyCA design or interconnection network and protocol. Each ASP has test benching for generating inputs and have been tested to generate the correct corresponding output, with additional time out team would have attempted to resolve this flaw. We found that if you “spam” the key to send a packet to configure the ASP it works, so we are unsure of the issue.

V. Conclusion

The aim of this project was to investigate and implement signal processing features for an oscilloscope based design to be run on a Hybrid Customisable Architecture (HyCA) and produce a ReCOP unit, 3 ASPs and 8 ports in the TDMA-MIN NoC, which allows connecting up to 8 cores. Additionally we were required to conduct signal processing on an incoming input waveform and outputs displayed via the seven segment display on the DE1-SoC board as well as generate analog waveforms to be outputted by the DAC. The DE1-SoC board will be used as an interface to send input packets via the switches and buttons located on the board to configure the ASP’s, ADC and DAC. Our team had successfully created the HyCA and displayed a working RECOP design for reading instruction memory and generating data packets.

Our system implements functionality for a simplified recop instruction set tailored specially to allow configuration and control of multiple connected data-processing application specific processors as well memory mapped IO control to the DE1-Soc board. This is achieved through communication between the control recop processor and the provided IONode where the values of the memory-mapped hex registers are assembled into a packet and sent across the TDMA-MIN to be displayed. The system supports up to 8 connected blocks through the TDMA-MIN where we currently utilize 7 of these connections; IONode, recop, ADC, DAC, ASP0, ASP1, ASP2 although we have only successfully been able to implement communication between the recop and the IONode, ADC and DAC. In future we would like to extend the current program to support data memory storage for more complex digital signal processing algorithms such as frequency detection; where individual DP-ASP functionality would also be extended. Improvements in the user IO such as LEDs indicating connections between individual processors would also improve usability of the system although we are limited in this case by the available hardware on the DE1-Soc board.

Appendix A

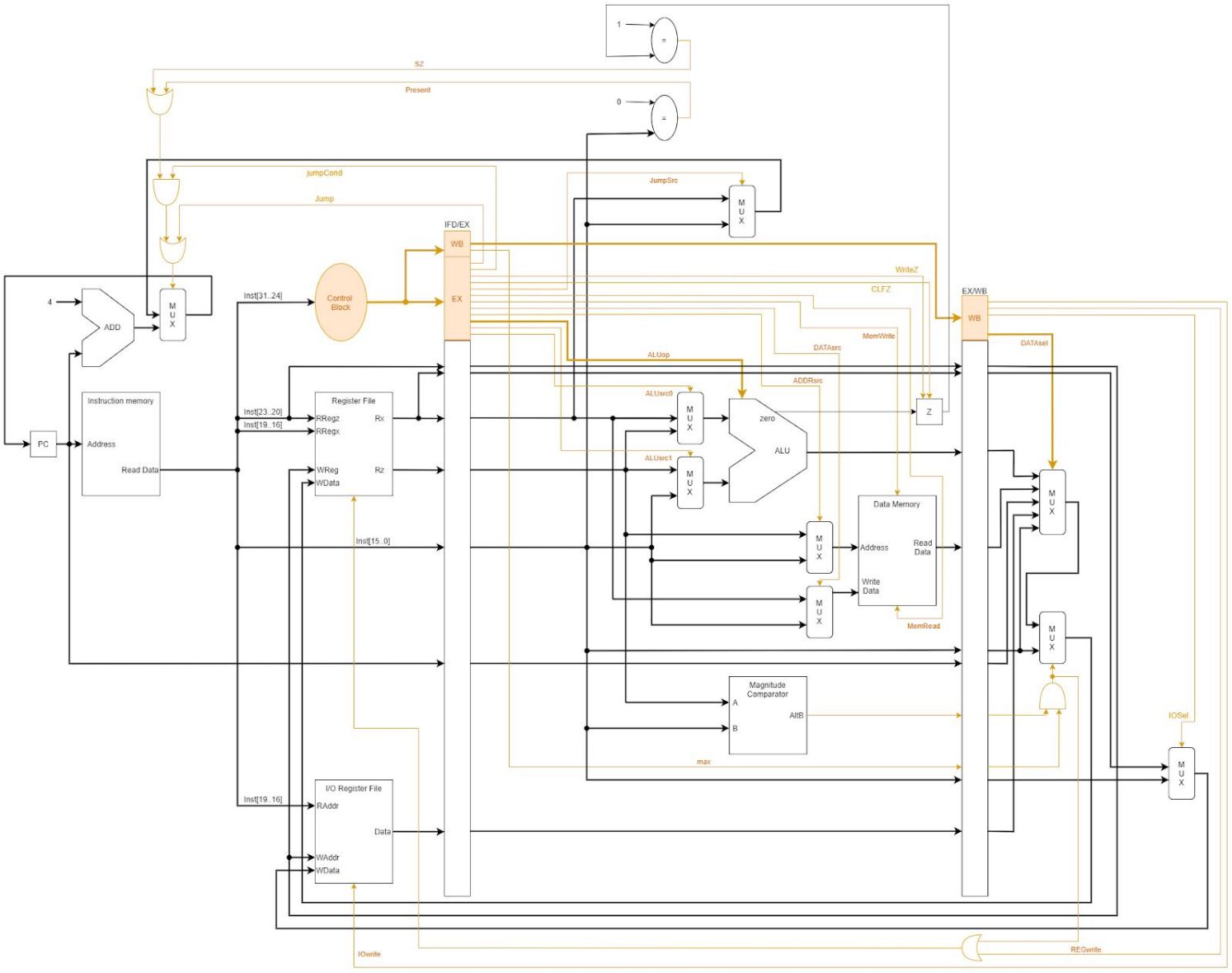

Figure 9. Pipeline Datapath