**BYOC course**

**Assignment #6**

**HW6 MIPS CPU**

**Appendices**

**Appendix A**

A.1) Fill up the following table describing what happens in each CK cycle in all instructions. You should specify the specific operations that are required for the execution of the instruction.

We filled in the Rtype and j instructions – as examples. We also gave the list of required registers & signals to be mentioned in the table, in the ori instruction line.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| phase | **IF** | **ID** | **EX** | **MEM** | **WB** |
| Instruction |
| Rtype | IR=IMem[PC]  PC= PC+4 | A=GPR[Rs]  B=GPR[Rt]  Active signals:  RegDst=’1’  RegWrite=’1’  ALUOP=”10”  MemToReg=’0’ | ALUOUT = A op B  Rd is chosen:  Rd\_pMEM=Rd\_pEX | ALUOUT\_pWB=  ALUOUT  (ALUOUT is delayed 1ck) | GPR[Rd\_pWB]  = ALUOUT\_pWB |
| addi | IR=IMem[PC]  PC= PC+4 | A=GPR[Rs]  sext\_imm  Active signals:  RegWrite <= '1';  RegDst <= '0';  ALUsrcB <= '1';  MemToReg <= '0';  MemWrite <= '0';  JAL <= '0';  ALUOP=”00” (add) | ALUOUT = A + sext\_imm\_reg  Sext\_imm\_reg = sext\_imm  Rd is chosen:  Rd\_pMEM=Rd\_pEX | ALUout\_reg = ALU\_output  (delay of 1 ck) | GPR\_wr\_data = ALUout\_reg\_pWB;  Rd\_pWB = Rd\_pMEM=Rt  GPR[Rt] = ALUout\_reg\_pWB |
| ori | Need to tell what is loaded to IR & PC – the relevant regs. | Again, all regs that are relevant (A, B, sext\_imm, PC in j & branch)  Also – all **active** signals created at the ID phase | All regs that are relevant (ALUOUT, B\_reg\_pMEM, Rd\_pMEM, sext\_imm) | All regs that are relevant (ALUOUT\_reg\_bWB, Rd\_pWB, MDR)  MDR= DMem[adrs ] or  DMem[adrs]=B\_reg\_pMEM | GPR[Rd\_pWB]  = ALUOUT\_pWB |
| lui | IR=IMem[PC]  PC= PC+4  Rs = “00000”(set when recognizing the opcode of lui) | A\_reg = 0x00000000  (set to zero)  sext\_imm <= imm(15 downto 0) & x"0000";  Active Signals:  RegWrite <= '1';  RegDst <= '0';  ALUsrcB <= '1';  MemToReg <= '0';  MemWrite <= '0';  JAL <= '0';  ALUOP=”00” (add) | sext\_imm\_reg = sext\_imm  ALUOUT = A + sext\_imm\_reg  Rd is chosen:  Rd\_pMEM=Rt\_pEX | ALUout\_reg = ALU\_output | GPR[Rd\_pWB] = aluout\_reg\_pWB  ALUout\_reg\_pWB = ALUout\_reg;  Rd\_pWB = Rd\_pMEM |
| beq | PC = PC +4  IR = IMem[PC] | Active Signals  ALUOP=”01”  If Rs\_equals\_rt =’1’ PC\_Source = “01”  Else  Pc\_source = “00”  Pc\_source =“00”  Then PC\_Plus\_4  PC\_source = “01”  Then branch\_adrs | - | - | - |
| bne | PC = PC +4  IR = IMem[PC] | ALUOP=”01”  The inverse of the above logic.  Pc\_source =“00”  Then PC\_Plus\_4  PC\_source = “01”  Then branch\_adrs | - | - | - |
| lw | PC = PC +4  IR = IMem[PC] | A= Rs  Sext\_imm = imm  Active Signals:  Aluop = “00”  Regwrite = ‘1’  Regdest =’0’  Alusrcb = ‘1’  Memwrite = ’0’  Memtoreg = ‘1’  JAL = ‘0’ | sext\_imm\_reg = sext\_imm  ALUout = A + sext\_imm\_reg  Rd is chosen:  Rd\_pMEM=Rt\_pEX | MDR= DMem[ALUout] | GPR\_wr\_data = MDR\_Reg (data from memory)  Rd\_pWB = Rd\_pMEM; |
| Sw | PC = PC +4  IR = IMem[PC] | A=Rs  B=Rt  sext\_imm = imm  Active Signals:  Aluop = “00”  Regwrite = ‘0’  Regdest =’0’  Alusrcb = ‘1’ Memtoreg = ‘0’  MemWrite = ‘1’  JAL = ‘0’ | sext\_imm\_reg = sext\_imm  ALUout = Rs (=A) + sext\_imm\_reg  B\_reg\_pMEM = Rt (=B) | DMEM[ALUout] = GPR[Rt] |  |
| j | IR=IMem[PC]  PC=PC+4 | PC= jump adrs | nothing | nothing | nothing |
| jal | IR=IMem[PC]  PC= PC+4 | PC = jump\_adrs  B = Rt = $31  Active Signals  ALUop = “00”  RegWrite = ‘1’  RegDst =’0’  ALUsrcB = ‘0’  MemToReg =”0”  MemWrite =”0”  JAL =’1’ | PC\_plus\_4\_pEx = PC\_plus\_4\_pID  Rd is chosen:  Rd\_pMEM=Rt\_pEX  JAL\_pEX = JAL | JAL\_pMEM = JAL\_pEX  PC\_plus\_4\_pMEM = PC\_plus\_4\_pEx | JAL\_pWB = JAL\_pMEM  Rd\_pWB = Rd\_pMEM  If JAL\_pWB = ‘1’ then  GPR\_wr\_data = PC\_plus\_4\_pWB  →  GPR[Rd\_pWB]= PC\_plus\_4\_pWB |
| jr | IR=IMem[PC]  PC= PC+4 | PC= jr\_adrs (=Rs)  Rd = $0 = 0 | - | - | - |

Answer the following questions.

A.2) Describe the changes done in order to support the ORI instruction.   
In order to prevent sign extension, we’ve modified the sign extension circuit in the fetch unit.

And added another case to address the ORI opcode -   
when b"001101" => sext\_imm <= x"0000" & imm(15 downto 0); -- ORI   
(this is Fuchsia! as you can clearly see).

Additionally, we’ve modified the ALU command control circuit and added the following case to support

the ORI instruction.  
when b"11" => ALU\_cmd <= b"001"; -- added OR when the command is ORI

Of course we’ve also modified the active control signals in the Decode phase to make sure the correct data paths are taken and the instruction is peformed correctly:

when b"001101" => -- ori

ALUOP <= b"11";

RegWrite <= '1';

RegDst <= '0';

ALUsrcB <= '1';

MemToReg <= '0';

MemWrite <= '0';

JAL <= '0';

A.3) Describe the changes done in order to support the LUI instruction.  
When executing the LUI instruction we write to the GPR file, therefore we’ve added the required changes to the active control signals selection in the decode phase. As follows:

when b"001111" => -- lui

ALUOP <= b"00";

RegWrite <= '1';

RegDst <= '0';

ALUsrcB <= '1';

MemToReg <= '0';

MemWrite <= '0';

JAL <= '0';

In addition, once the LUI instruction is executed, we automatically set the Rs value to the zero register – since it always contains 32 zero bits, in the Ex phase:

process(CK,RESET,Opcode)

begin

if RESET='1' then

A\_reg <= x"00000000";

elsif CK'event and CK = '1' and HOLD='0' then

if Opcode = b"001111" then -- case of LUI

-- Setting A\_reg to be the 0

A\_reg <= x"00000000";

else

A\_reg <= GPR\_rd\_data1;

end if;

end if;

end process;

We chose to set A\_reg to 0 to make sure that the value taken from the GPR contains 32 bit of zeros (logical zero).

Also, to finalize the implementation of the LUI instruction, we’ve modified the sign extension circuit to support the shifting of 16 bits when detecting the opcode of the LUI instruction. As can be seen below (you will definitely enjoy this ! )

When (=Opcode) b"001111" => sext\_imm <= imm(15 downto 0) & x"0000"; --LUI

A.4) Describe the changes done in order to support the JR instruction.  
Supporting this command was pretty easy, as a wise man once said (in the course bible).

We’ve changed the constant value previously assigned to the “jr\_adrs” to to be the “jr\_adrs\_in” coming from the “Top”, in the fetch unit:

jr\_adrs <= jr\_adrs\_in;

In addition, we’ve connected the Jr\_address in the “Top” (connecting to the jr\_adrs\_in in the fetch unit) to receive the GPR\_rd\_data1\_wt\_fwd – in order to supply the fetch unit with the relevant data once the Jr instruction has been executed.

jr\_address <= GPR\_rd\_data1\_wt\_fwd ; -- HW6 change to support fwding for Jr

In the fetch unit, we’ve modified the PC Source decoder, to detect the Jr instruction, and select the proper PC source, as can be seen below. (Selecting “10” will select the jr\_adress given from the “Top”). To achieve this we’ve addressed the “funct” bits of the command, as can be seen below:

when b"000000" =>

if funct = b"001000" then

PC\_source <= b"10"; -- jr

The above change will ensure we will take the correct data from the Jr\_address\_in and put it into the PC Source.

A.5) Describe the changes done in order to support the JAL instruction.

Supporting this command, phew, was very involving and … we felt immersed in the command.

BUT! In the end we did it.

First, we’ve modified the ID active control signal circuit to enable support of the JAL instruction. This means adding the JAL signal.

when b"000011" => -- JAL

ALUOP <= "00";

RegWrite <= '1'; -- a must to write to the GPR of return address

RegDst <= '0';

ALUsrcB <= '0';

MemToReg <= '0';

MemWrite <= '0';

JAL <= '1';

Inside the fetch unit, we’ve modified the PC Source decoder to select the “11” value which will ensure that we will use the jump address as the input to the PC source. (the address is given from the “Top”).

when b"000011" => PC\_source <= b"11"; --jal

As part of the JAL instruction we make sure that we write the PC\_Plus\_4 to register $31 in the GPR file.

To do so, we set the Rt value to “11111” (31 in binary!).

process(Opcode,IR\_reg)

begin

if Opcode = b"000011" then -- jal

Rt <= b"11111"; -- JAL support = to enable writing PC+4 to register 31

else

Rt <= IR\_reg(20 downto 16);

end if;

end process;

This is done by propagating both the address of the PC+4 to the GPR and the JAL instruction control signal until we reach the WB phase.

To propagate, as we’ve done in previous exercises we use registers that move the JAL instruction control signal value each ck cycle (each phase). Upon reaching the shores of the WB phase, we insert through the MemToReg mux the PC\_Plus\_4\_pWB value in to the GPR\_wr\_data. This will allow the inserting the value of the PC+4 into the GPR (via the GPR\_wr\_data variable).

--MemToReg mux --@@@HW6 requires changes to support JAL instruction

process(MemToReg\_pWB,MDR\_reg,ALUOut\_reg\_pWB,JAL\_pWB,PC\_Plus\_4\_pWB)

begin

if JAL\_pWB = '1' then

GPR\_wr\_data <= PC\_Plus\_4\_pWB;

Rt is forced to be $31 since we want to ensure that RegDst mux will choose the Rd\_pMEM to be Rt\_pEX (setting Rt\_pEX to Rd\_pMEM).

**Appendix B**

Answer the following questions.

B.1) What are the limitations due to the pipeline latency of the following combinations:

* lw after add where the add Rd is the lw Rs
* lw after add where the add Rd is the lw Rt
* add after lw where the lw Rt is the add Rt
* beq after lw where the lw Rt is the beq Rs

Use a similar figure to Fig.2 and Fig. 3 to demonstrate your answers. Explain your answer!

B.1.a - lw after add where the add Rd is the lw Rs [ e.g., lw $4,16($3) ]

WB

EX

ID

IF

WB

EX

ID

IF

WB

EX

ID

IF

Nop

Nop

add **$3**,$5,$8

WB

EX

ID

IF

Nop

WB

EX

ID

lw Rt offset($3)

IF

CK

MEM

MEM

MEM

MEM

MEM

EX

Answer:  
Without data forwarding, the pipeline has inherent latency. In the case of an LW instruction after an ADD operation where the instruction uses the same register that the previous operation was about to write to, we will use an obsolete value, unless we wait until the add WB phase is over, only then can we access the GPR and obtain the correct Rs value (for thr Rt) since it has been written (the ID phase of the lw).

B.1.b - lw after add where the add Rd is the lw Rt

WB

EX

ID

IF

WB

EX

ID

IF

WB

EX

ID

IF

lw $3 offset(Rs)

add **$3**,$5,$8

WB

EX

ID

IF

WB

EX

ID

IF

CK

MEM

MEM

MEM

MEM

MEM

EX

Answer: *There is no limitation in the sequence of given instructions*

The infor mation stored in the first add operation wil be overwritten by the information written in the lw instruction. (Since they both write to the same register). The limitation imposed here is not due to pipeline latency but due to the logic of the instructions (assuming this is intentional)

B.1.c - add after lw where the lw Rt is the add Rt

CK

WB

EX

ID

IF

lw **$3**,16($10)

MEM

WB

EX

ID

IF

nop

MEM

WB

EX

ID

IF

MEM

nop

WB

EX

ID

IF

MEM

nop

WB

EX

ID

IF

EX

add Rd, Rs, $3

MEM

Lw GPR[Rt] <- DMEM[GPR[$10]+16]

Add GPR[Rd] <- GPR[Rs] + GPR[Rt]

Answer:

The result of the LW is first written to the GPR in the WB phase of the intstruction, to the Rt register. The add operation requires the result in its ID phase(only then is the data actually already written to the GPR), therefore we must wait as described above – 3 ck cycles.

B.1.d - beq after lw where the lw Rt is the beq Rs

WB

EX

ID

IF

WB

EX

ID

IF

WB

EX

ID

IF

nop

nop

lw **$3**,16($10)

WB

EX

ID

IF

nop

WB

EX

ID

beq $3 offset(rs)

IF

CK

MEM

MEM

MEM

MEM

MEM

EX

Answer:

Lw GPR[Rt] <- DMEM[GPR[$10]+16]

Beq if GPR[Rt]=GPR[Rs] → offset (branch offset)

As in previous questions, we must wait until the information is stored into the Rt register in the GPR (the WB stage of the lw operation) and only then, can we access the information correctly in the ID stage of the beq instruction. (as can be seen above) Therefore, we must wait for 3 ck cycles.

B.2) What are the limitations of all cases of B.1 after you add the Data Forwarding? . Explain your answer!

B.2.a - lw after add where the add Rd is the lw Rs

WB

EX

ID

IF

WB

EX

ID

IF

WB

EX

ID

IF

I : lw Rt offset($3)

II : lw Rt offsfet($3)

add **$3**,$5,$8

WB

EX

ID

IF

III : lw Rt offset($3)

WB

EX

ID

IV : lw Rt offset($3)

IF

CK

MEM

MEM

MEM

MEM

MEM

EX

Answer:

Add GPR[$3] = GPR[$5]+GPR[$8]

Lw GPR[Rt] = DMEM[GPR[$3]+offset]

Case I : Data forwading enalbes using the result of the add operation from the ALU output into the execute phase of the next instruction. Thus, allowing the lw operation to be peformed without latency of the pipeline. We handle this case by comparing the Rd\_pMEM to the Rs\_pEX, between instructions, if equal we take the result of the previous ALU operation.

Case II : No latency issues either, thanks to the data forwarding we can use the data written through the memtoreg MUX result, since forwarding allows this transfer of information instead of the register value stored in the GPR when the lw reaches the EX phase.

Case III: No latency issues either ! The lw will reach the ID phase, and the information stored in the GPR\_wr\_data will be used when reaching the ID phase (instead of the regular a\_reg source from the GPR).

 IV: Obviously after the WB phase there is no latency issue

B.2.b - lw after add where the add Rd is the lw Rt

WB

EX

ID

IF

WB

EX

ID

IF

WB

EX

ID

IF

lw $3 offset(Rs)

add **$3**,$5,$8

WB

EX

ID

IF

WB

EX

ID

IF

CK

MEM

MEM

MEM

MEM

MEM

EX

Answer:

There is no limitation here, just like the previous question (without forwarding). Since we will overwrite the $3 register either way, regardless of the timing of the instruction. (so we have no problem with the data loaded into the $3 in the add WB and the data written in the WB phase of the LW (or ID phase).

B.2.c - add after lw where the lw Rt is the add Rt

WB

EX

ID

IF

WB

EX

ID

IF

WB

EX

ID

IF

nop

add Rd Rs $3

lw **$3**,16($10)

WB

EX

ID

IF

WB

EX

ID

IF

CK

MEM

MEM

MEM

MEM

MEM

EX

Answer

In order to achieve the correct result, we must wait 1 Ck cycle and wait for the WB phase to use the value of $3 in the add instruction (This is supported by the data forwarding circuit in the EX phase of the add).

For usage after the above drawing, we can execute the add instruction at will any ck cycle afterwards thanks to the support of forwarding information from the transparent GPR in the ID phase.   
\* in the last add – 3 clock cycles afterwards – we will already have the new information written after the WB phase.

B.2.d - beq after lw where the lw Rt is the beq Rs

WB

EX

ID

IF

WB

EX

ID

IF

WB

EX

ID

IF

NOP

NOP

lw **$3**,16($10)

WB

EX

ID

IF

beq $3 offset(rs)

WB

EX

ID

IF

CK

MEM

MEM

MEM

MEM

MEM

EX

Answer

Data forwarding enables the result of the lw instruction (written into the Rt register) to be available to the beq instruction only after waiting 2 Ck cycles and only then running the beq instruction.

We cannot perform the instruction earlier as the information must be written in to the GPR from the memory in the lw WB.

This is thanks to transparent GPR, in ID phase of the beq the information from the WB is already available. (see above drawing).

\*Of course no limitation exists if we execute the beq instruction 4 Ck cycles afterwards

B.3) How many times do we perform the instruction following a jal instruction? Explain in detail. What are the implications? If this is a problem, what do you suggest in order to solve it?

We perform the instruction following the JAL instruction twice – this is because until we jump we still fetch,decode and run the following instruction in the pipeline. When returning to the address written into the $31, it stores the PC+4, which is exactly the instruction following the JAL instruction. In order to avoid performing the same instruction twice, we can place a nop instruction immediately after the JAL instruction.

B.4) How soon after jal instruction can we issue a jr $31 instruction in order to return to the right location in the code? Give the answer before data forwarding is added and then after the data forwarding is added. . Explain your answer!

No data forwarding:

With no data forwarding, we must wait until the register $31 is written with the contents of the PC\_plus\_4 data. This is due to the fact that we must wait until the WB phase of the JAL routine to store the value propagated by the JAL instruction.

WB

EX

ID

IF

WB

EX

ID

IF

WB

EX

ID

IF

jal routine1

WB

EX

ID

IF

WB

EX

ID

Jr $31

IF

CK

MEM

MEM

MEM

MEM

MEM

EX

With data forwarding:

When performing a Jr instruction, we must have the $31 register already available to us in the ID phase of the Jr instruction. This imposes a limitation as we must wait 2 ck cycle after the JAL instruction for the information to be available. During the WB phase of the JAL routine – we will be able to access the output of the data that will be written to the GPR through the transparent GPR and therefore – will be able to execute the Jr $31 instruction.

WB

EX

ID

IF

WB

EX

ID

IF

WB

EX

ID

IF

nop

nop

jal routine1

WB

EX

ID

IF

Jr $31

WB

EX

ID

IF

CK

MEM

MEM

MEM

MEM

MEM

EX

**Appendix C**

Answer the following questions.

C.1) What are the limitations due to the pipeline latency of the following combinations (assume Data Forwarding already exists):

* beq after add where the add Rd is the beq Rt
* beq after lw where the lw Rt is the beq Rs

Use a similar figure to Fig.2 and Fig. 3 to demonstrate your answers. Explain your answers!

C.1.a - beq after add where the add Rd is the beq Rt

WB

EX

ID

IF

WB

EX

ID

IF

WB

EX

ID

IF

nop

nop

add **$3**,$5,$8

WB

EX

ID

IF

Beq Rs offset($3)

WB

EX

ID

IF

CK

MEM

MEM

MEM

MEM

MEM

EX

Since DF exists, we must wait 2 ck cycles after the add instruction in order to have the result of the add operation in the mem phase available during the ID phase of the beq instruction. The data will be forwarded from the transparent GPR, during the ID phase of the beq instruction.

C.1.b - beq after lw where the lw Rt is the beq Rs

WB

EX

ID

IF

WB

EX

ID

IF

WB

EX

ID

IF

nop

nop

lw **$3**,16($10)

WB

EX

ID

IF

Beq rs $3 offset

WB

EX

ID

IF

CK

MEM

MEM

MEM

MEM

MEM

EX

The answer from the previous question remains the same as it applies to both instructions, the same considerations apply:  
When the lw runs it stores the information from the DMEM into the GPR $3 address, this information will then be written during the WB phase of the lw instruction.

When running the beq instruction, to correctly access the newly retrieved information from the DMEM, we must wait 2 ck cycles after the lw and use the transparent GPR feature to access the information during the ID phase of the beq instruction.

C.2) What are the limitations of all cases of C.1 after you add the Branch Forwarding? . Explain your answers!

C.2.a - beq after add where the add Rd is the beq Rt

WB

EX

ID

IF

WB

EX

ID

IF

WB

EX

ID

IF

Beq Rs offset($3)

add **$3**,$5,$8

WB

EX

ID

IF

WB

EX

ID

IF

CK

MEM

MEM

MEM

MEM

MEM

EX

nop

Adding the branch forwarding allows to shorten the ck cycle delay by 1 instruction thanks to the modifications we’ve made – once the beq instruction reaches the ID phase where it loads the values for Rt and Rs the add reaches the MEM phase and the branch forwarding allows to extract the value from ALUout\_reg into the beq Rt value.

C.2.b - beq after lw where the lw Rt is the beq Rs

WB

EX

ID

IF

WB

EX

ID

IF

WB

EX

ID

IF

Nop

Nop

lw **$3**,16($10)

WB

EX

ID

IF

Beq $3 offset(Rt)

WB

EX

ID

IF

CK

MEM

MEM

MEM

MEM

MEM

EX

Since the beq instruction will run after the lw, we must wait until the data is retrieved from the DMEM, only then will we be able to utilize it in the beq instruction. This mens that we will have to wait 2 ck cycles after the lw instruction in order to use the information as the beq Rs value (using data forwarding) –therefore the limitation from before has not been improved thanks to branch forwarding. This is possible thanks to the transparent GPR.

C.3) Why can’t we check the result of the previous instruction (time slot n-1) by a beq instruction following it (time slot n)?

Even with data forwarding and branch forwarding, the previous instruction before the beq instruction will reach the EX stage and will not finish the calculation in the ALU unit to utilize in the beq instruction (since we will want some information that is to be stored in the previous instruction – e.g. Rd in an add operation will be come Rs or Rt in the beq). Therefore we will have to wait an additional ck cycle.

C.4) List all of the limitations for Assembly programmer you can think of that still exist after adding the Data & Branch Forwarding circuits. . Explain your answer!

1. As we’ve seen we still must always wait in all Rtype instructions or I-type instructions for the ALU to finish at the minimum to use the result of the calculations in following instructions (wait 1 ck cycle).

2. (after solving the next question) – The shortest loop is always 2 instructions, because of the design of the MIPS pipeline.

3. In order to fully load 32 bits we must use 2 instructions LUI and ORI in combination for example.

4. You may not use add after an lw operation utilizing the same register (immediately after) – as seen above.

C5) What is the shortest loop code possible (not an infinite loop)? Any limitations? Explain in detail

The shortest instruction require – always – 2 stages at the least (2 ck phases) –

The IF and the ID. Instructions that create loops are J to the PC register itself – we will fetch it and after the ID phase we will jump to the same address we started from.

0x400000 : J 0x400000

0x400004 : nop (this instruction is required since until we jump we will already fetch and decode the nop)

This loop will cause the program to jump back to 0x400000 and run infinitely. 2 instructions are mandatory even though we will not do anything with second instructions, because of the design of the pipeline.

The shortest possible loop, as explained, consists of 2 instructions. If we modify or create a new jump instruction which injects into the IR a nop instruction during the ID phase – we will shorten the shortest loop command to a single instruction.