Pipelining (Modified Lab 7 part 3):

Here, in EX1 also we have an ADD4 unit. No SUB3 at all. One can perform ADD4 in each of the two stages EX1 and EX2 to do ADD8 as shown below.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Opcode</th>
<th>MSD</th>
<th>32-bit instruction in hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td></td>
<td>0 0 0 0 0 0</td>
<td>0</td>
<td>000000DS</td>
</tr>
<tr>
<td>MOV</td>
<td>SR, SX; (SR) &lt;= (SX)</td>
<td>1 0 0 0 8</td>
<td>8</td>
<td>800000DS</td>
</tr>
<tr>
<td>SUB3</td>
<td>SR, SX; (SR) &lt;= (SX) + 3</td>
<td>0 1 0 0 4</td>
<td>4</td>
<td>400000DS</td>
</tr>
<tr>
<td>ADD4</td>
<td>SR, SX; (SR) &lt;= (SX) + 4</td>
<td>0 0 1 0 2</td>
<td>2</td>
<td>200000DS</td>
</tr>
<tr>
<td>ADD8</td>
<td>SR, SX; (SR) &lt;= (SX) + 8</td>
<td>0 0 0 1 1</td>
<td>1</td>
<td>100000DS</td>
</tr>
</tbody>
</table>

An ADD4 instruction can now execute from either EX1 or EX2.
ADD4 tries to execute as soon as possible so that he can provide forwarding help to his juniors. However, he will not insist on executing from EX1, if he himself is dependent on, say, an ADD8 ahead of him. In that case he will skip EX1 (SKIP1 = 1) and execute from EX2. Let us call him, "ADD4_Skpd1" (ADD4 Skipped EX1). Now if the next ADD4 is dependent on this ADD4_Skpd1, then he will also skip EX1 and will become another ADD4_Skpd1! Now if ADD8 comes after him he needs to stall himself as he can not get help from this ADD4_Skpd1 in time.

It is important to know if the ADD4, currently standing in EX2 has finished execution already in EX1 (and hence may activate SKIP2 here) or skipped execution in EX1 and came here to get forwarding help and then perform the addition of 4 here in EX2. To this end, Mr. Trojan carried the SKIP1 signal into EX2 using a FF in the EX1/EX2 stage register. This signal is called EX2_SKIP1 (SKIP1 carried into EX2).

Do not forget the MOV instruction. Let us call the MOV instruction, a "she". She necessarily skips both EX1 and EX2 and may receive forwarding help in either EX1 or EX2 or perhaps in both EX1 and EX2. Circle all correct statements below.

A. There are occasions where she has to receive the needed forwarding help only in EX1.
B. There are occasions where she has to receive the needed forwarding help only in EX2.
C. There is no harm if she receives help in EX1 from EX2 occupied by ADD8 or ADD4_Skpd1, as this wasteful help will be replaced by correct help in the next clock.
D. Unlike ADD4 or ADD8, she will never regret (feel sorry) for receiving help in EX2.
E. In short, she can receive help in EX1 as well as EX2 without any worry.

You never need to stall (circle all correct answers): a NOP, a MOV, an ADD4, an ADD8
1. Complete all missing connections to the Reg. File.
   Also complete the RA(Result Address) connection in ID stage (ID_RA).
2. Complete all five enable (EN) controls on the pipeline registers (including PC).
3. Complete the forwarding path from EX2 to EX1. Should it start from upstream or downstream of the X2_max?
4. Draw the logic to produce STALL, PRIORITY, FORW1, SKIP1, FORW2, SKIP2 on the next page.

Modified LAB 7 Part 3 Block Diagram

Q#1
1.1 Draw logic to produce all the six signals and complete the rest of the design on the previous page. It may be a good idea to finish Q1.2 before completing this.

![Logic Diagram]

If nearest senior wants to help, we give him priority. Otherwise, senior 2 info from WB is brought through the priority mux upto the forwarding mux, without even checking to see if he is qualified to help. We could do this because, we will reject his help if it is inappropriate at the forwarding mux. So forwarding mux takes help that is coming via the priority mux, if any of the seniors were to help.

This EX2-Skip qualification, though sounds meaningful, is actually not necessary, as the instruction looking for help from such senior would have been stalled in ID stage itself.
If I am an ADD4 in EX1, I would like to finish my computation here, but what if I am dependent on an immediate; senior, who is
1. an ADD8
2. an ADD4 who had skipped in his EX1
Then I have to SKIP EX1 and postpone my operation to EX2 MOV, of course skips both.

We can accept help from a senior in WB into EX2 provided there is a dependency and the senior is not a NOP. However, we should reject this help, the second time, if we (the recipient) are an ADD8 instr or an ADD4 who did not skip EX1. MOV can receive help twice with no negative effect.

MOV will definitely skip EX2.
ADD4 will skip EX2, if he did not skip EX1.
Another Solution for FORW2

We can accept help from a senior in WB into EX2 provided there is a dependency and the senior is not a NOP. The ADD4 who skipped ADD4 operation in EX1, can come to EX2 to seek help. Similarly a MOV can get help here.
1.2

In the following section, the word "interdependent" is used to mean "each instruction is dependent on its immediate senior instruction".

A series of interdependent ADD8 instructions, causes each dependent ADD8 to stall for ______ (0/1/2) clocks.

A series of interdependent ADD4 instructions, causes each dependent ADD4 to stall for ______ (0/1/2) clocks.

A series of interdependent instructions alternating between ADD8 and ADD4 and starting with ADD8 (example ADD8, ADD4, ADD8, ADD4, ...), causes each dependent ADD8 to stall for ______ (0/1/2) clocks and each dependent ADD4 to stall for ______ (0/1/2) clocks.

No instruction after a MOV instruction needs to stall. True / False

If a program has millions of instructions with no ADD8 at all, the best CPI is ______. This "best" CPI is actually the only CPI (whether you call it best or worst or constant). True / False

The "worst" CPI for any program (may be with a sequence of all interdependent ADD8’s) is ______.

2

(15 + 18 + 18 = 51 points) 30 min.

5-stage early branch pipeline (see diagram on next page) -- avoid spurious stall for SW

The current HDU (not HDU_Br) in the early branch pipeline causes a few spurious stalls as it does not check to see if the instruction being stalled in ID stage is really a register reading instruction.

Miss Trojan has redesigned the HDU to avoid any spurious stalls. The control unit informs the HDU if the instruction is a 2 or 1 or 0 source-register-reading instruction.

Miss Trojan further improved the HDU for SW instruction (a 2-source instruction) by observing that SW can wait to receive forwarding help for one of its source registers as late as in MEM stage.

You explore her idea and draw a forwarding mux on the next page.

Consider the following two situations.

\[
\begin{array}{c}
\text{#A} \\
\text{lw} \quad 8, \quad 40 \quad ($2); \\
\text{sw} \quad 8, \quad 60 \quad ($2); \\
\end{array}
\]

\[
\begin{array}{c}
\text{#B} \\
\text{lw} \quad 8, \quad 40 \quad ($2); \\
\text{sw} \quad 2, \quad 60 \quad ($8); \\
\end{array}
\]

In situation #A

(i) you need to stall if we use the original HDU  T / F
(ii) you need to stall if we use the new HDU  T / F
(iii) if we use the original HDU, you forward $8 \text{ from the lw when it is in } \underline{\text{WB}} \text{ (EX / MEM / WB) to sw in } \underline{\text{EX}} \text{ (ID / EX / MEM).}
(iv) if we use the new HDU and Miss Trojan’s forwarding, you forward $8 \text{ from the lw when it is in (EX / MEM / WB) to sw in } \underline{\text{MEM}} \text{ (ID / EX / MEM).}
(v) there is no dependency problem associated with $2  T / F

In situation #B (answer the same questions)

(i) you need to stall if we use the original HDU  T / F
(ii) you need to stall if we use the new HDU  T / F
(iii) if we use the original HDU, you forward $8 \text{ from the lw when it is in } \underline{\text{WB}} \text{ (EX / MEM / WB) to sw in } \underline{\text{EX}} \text{ (ID / EX / MEM).}
(iv) if we use the new HDU and Miss Trojan’s forwarding, you forward $8 \text{ from the lw when it is in } \underline{\text{WB}} \text{ (EX / MEM / WB) to sw in } \underline{\text{EX}} \text{ (ID / EX / MEM).}
(v) there is no dependency problem associated with $2  T / F
Detailed implementation of Early Branch suggested in 3rd Ed.

Draw the new forwarding mux, and FF to carry mux selec cotol signal
The forwarding mux (being added by you on the previous page) is controlled by \texttt{MEM\_FORW\_SW} (for Forward for SW in MEM stage) control signal. This is actually generated in the FU (forwarding unit) in the EX stage and is called \texttt{EX\_FORW\_SW}, and is carried into the MEM stage (through a FF in the EX/MEM stage register) to become \texttt{MEM\_FORW\_SW}.

Generate \texttt{EX\_FORW\_SW} in the FU by modifying its pseudo code below.

\textbf{FU (the original Forwarding Unit in EX stage):}

1. **EX Hazard:**

   \begin{verbatim}
   if [ 
   \hspace{1cm} \text{EX/MEM.RegWrite} \\
   \hspace{1.5cm} \text{and (EX/MEM.WriteRegister} \neq \text{0}) \\
   \hspace{2cm} \text{and (EX/MEM.WriteRegister} == \text{ID/EX.ReadRegister_RS}) \\
   \] 
   \hspace{1cm} \text{then make FW_RS_MEM} = \text{1}
   \end{verbatim}

\begin{verbatim}
if [ 
\hspace{1cm} \text{EX/MEM.RegWrite} \\
\hspace{1.5cm} \text{and (EX/MEM.WriteRegister} \neq \text{0}) \\
\hspace{2cm} \text{and (EX/MEM.WriteRegister} == \text{ID/EX.ReadRegister_RT}) \\
\] 
\hspace{1cm} \text{then make FW_RT_MEM} = \text{1}
\end{verbatim}

\textbf{2. MEM Hazard:}

   \begin{verbatim}
   if [ 
   \hspace{1cm} \text{MEM/WB.RegWrite} \\
   \hspace{1.5cm} \text{and (MEM/WB.WriteRegister} \neq \text{0}) \\
   \hspace{2cm} \text{and (MEM/WB.WriteRegister} == \text{ID/EX.ReadRegister_RS}) \\
   \] 
   \hspace{1cm} \text{then make FW_RS_WB} = \text{1}
   \end{verbatim}

\begin{verbatim}
if [ 
\hspace{1cm} \text{MEM/WB.RegWrite} \\
\hspace{1.5cm} \text{and (MEM/WB.WriteRegister} \neq \text{0}) \\
\hspace{2cm} \text{and (MEM/WB.WriteRegister} == \text{ID/EX.ReadRegister_RT}) \\
\] 
\hspace{1cm} \text{then make FW_RT_WB} = \text{1}
\end{verbatim}
The forwarding mux (being added by you on the previous page) is controlled by \texttt{MEM\_FORW\_SW} (for Forward for SW in MEM stage) control signal. This is actually generated in the FU (forwarding unit) in the EX stage and is called \texttt{EX\_FORW\_SW}, and is carried into the MEM stage (through a FF in the EX/MEM stage register) to become \texttt{MEM\_FORW\_SW}.

Generate \texttt{EX\_FORW\_SW} in the FU by modifying its pseudo code below.

\begin{verbatim}
FU (the original Forwarding Unit in EX stage):

1. EX Hazard:

   if [ 
     EX/MEM.RegWrite
     and (EX/MEM.WriteRegister /= 0)
     and (EX/MEM.WriteRegister == ID/EX.ReadRegister_RS)
   ]
   then make FW_RS_MEM = 1

   if [ 
     EX/MEM.RegWrite
     and (EX/MEM.WriteRegister /= 0)
     and (EX/MEM.WriteRegister == ID/EX.ReadRegister_RT)
   ]
   then make FW_RT_MEM = 1

2. MEM Hazard:

   if [ 
     MEM/WB.RegWrite
     and (MEM/WB.WriteRegister /= 0)
     and (MEM/WB.WriteRegister == ID/EX.ReadRegister_RS)
   ]
   then make FW_RS_WB = 1

   if [ 
     MEM/WB.RegWrite
     and (MEM/WB.WriteRegister /= 0)
     and (MEM/WB.WriteRegister == ID/EX.ReadRegister_RT)
   ]
   then make FW_RT_WB = 1
\end{verbatim}

---

Since we are helping SW for its \texttt{rt} from an immediate \texttt{jump} load \texttt{word}, several paragraphs of the pseudo code remain unmodified.
3  \((6+10+24+5=45 \text{ points})\) 20 min.

Single Cycle CPU Datapath under the control of a 10-state control unit resembling the multi-cycle CPU control unit of the 1\textsuperscript{st} edition:

This new control unit is \textit{less of controlling} and more of \textit{apportioning} the needed clocks for fetching, decoding and executing.

Notice that this question is similar to the lab 7 P3 question reproduced on the side for your reference.

On the next to next page, we have reproduced the single cycle DPU with the single modification: the PC now has a PCWrite control like in the multicycle CPU. This is needed as we \textbf{do not} (do / do not) want to update PC more than once per instruction. This \textbf{includes} (includes / excludes) \texttt{beq} and \texttt{j} (jump) instructions, which, in the multi-cycle CPU of the 1\textsuperscript{st} edition, required PC to be updated \textbf{2} \((0/1/2/3)\) time(s) in total during the Fetch-Decode_Execute operation. \textit{Note: PC is incremented at the end of state 5 and again at the end of state 8, state 9.}

Mr. Bruin said, "After fetching the instruction in state S0 and incrementing PC, in each of the 9 remaining states, let us write the following:

\begin{center}
the 10 control signals, as per the table below as generated by the combinational CU of the single-cycle CPU
\end{center}

Then, depending on the instruction and its opcode, appropriate signals get activated, and the needed execution takes place, \textit{but in just needed number of short (10ns) clocks, instead of one long 40ns clock}.”

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|}
\hline
\textbf{Instruction} & \textbf{RegDst} & \textbf{ALU/Sec} & \textbf{MemToReg} & \textbf{RegWrite} & \textbf{MemRead} & \textbf{MemWrite} & \textbf{Branch} & \textbf{ALUOp1} & \textbf{ALUOp0} & \textbf{Jump} \\
\hline
\textbf{R-format} & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
\hline
\textbf{lw} & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
\hline
\textbf{sw} & X & 1 & X & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
\hline
\textbf{beq} & X & X & X & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
\hline
\end{tabular}
\end{table}

You immediately point out to him that (i) there \textbf{isn’t} (is, isn’t) any IR, hence he \textbf{shouldn’t} (should / shouldn’t) be incrementing PC in S0, (ii) by activating RegWrite in states 6 as well as 7, he would be writing \textbf{multiple times} (one time / multiple times) to the register file leading to \textbf{wrong} (right/wrong) results, for example, in \texttt{addu} \$2, \$2, \$2. (iii) for SW instruction he \underline{would} (would/ wouldn’t) be writing to the memory at the end of the state 2 potentially corrupting the memory as address is being calculated in state 2.
Your design: You divide the 10 control signals into two groups:

Less significant five: RegDst, MemtoReg, ALUSrc, ALUOp1, ALUOp0

More significant five: RegWrite, MemRead, MemWrite, Branch, Jump

First group of less significant five: You allow Bruin’s method. You allow these five signals to be generated by the combinational control unit of the single cycle CPU.

Second group of more significant five: To these you add PCWrite and you list them as appropriate in the 8-states (S2 through S9) below. Go ahead and complete the 8 states S2 through S9 below.

This design running at 10ns clock compared to the original single-cycle CPU running at 40ns is expected to be 4 times better. True / False

If a program consists of mainly load words, this design is expected to be __________________ compared to the original single-cycle CPU.

(superior / inferior)
Single Cycle CPU (supporting jump)
Note: PCWrite has been added.
4 \hspace{1cm} (12 + 3 = 15 \text{ points}) 6 \text{ min.}

Flushing by a successful branch:

Our Lab 6 Verilog code may have chosen to set or clear the "wrist-band" Flip-Flop (flush bit) in the stage register IF/ID on system reset using the \texttt{RESET} signal. Accordingly we discussed in class a solution for the Lab 6 Part 4 question on flushing two stages of the 7-stage pipeline.

For this question, let us assume that each designer is allowed to choose to set or clear each "wrist-band" Flip-Flop. He can choose to set one FF and clear another. Also some designers below assumed one delay slot where as some assumed zero delay slots. All 6 designs are correct based on their assumptions. Fill-in the table telling us what assumptions make the designs correct.

All control units are identical and are as per the textbook design (a 1-input means it is an instruction destined to be flushed).

Fill-in the table ===>
Write Clear or Set under FF1 and/or FF2 and write 0 or 1 under Delay Slot.

<table>
<thead>
<tr>
<th>Design #</th>
<th>IF1/IF2</th>
<th>FF1</th>
<th>FF2</th>
<th>ID</th>
<th>Delay Slot</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SET</td>
<td>SET</td>
<td></td>
<td>FF1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>CLEAR</td>
<td>SET</td>
<td></td>
<td>FF2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>CLEAR</td>
<td>SET</td>
<td></td>
<td>FF2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>SET</td>
<td>CLEAR</td>
<td></td>
<td>FF1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>CLEAR</td>
<td>SET</td>
<td></td>
<td>FF1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>CLEAR</td>
<td>CLEAR</td>
<td></td>
<td>FF2</td>
<td></td>
</tr>
</tbody>
</table>
Cache mapping techniques:

Fill-in all missing information in the table below based on information provided. In all four cases, it is the same amount of cache differently organized.

<table>
<thead>
<tr>
<th>Addr Space Size</th>
<th>Cache Size</th>
<th>Block Size</th>
<th>Mapping Technique</th>
<th>TAG FIELD</th>
<th>BLOCK OR SET FIELD (as appropriate)</th>
<th>WORD FIELD</th>
<th>BYTE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte addressable Processor</td>
<td>4 bytes</td>
<td>128 KB</td>
<td>Direct</td>
<td>( A_{31} - A_{17} )</td>
<td>2 ( A_{16} )</td>
<td>( A_{4} - A_{2} )</td>
<td>( A_{1} )</td>
</tr>
<tr>
<td>32-bit Data address</td>
<td>8 bytes</td>
<td>2 ( A_{32} )</td>
<td>Fully Associative</td>
<td>( A_{31} - A_{5} )</td>
<td>2 ( A_{4} ) ( \times ) ( 17 )</td>
<td>( A_{4} - A_{2} )</td>
<td>( A_{1} )</td>
</tr>
<tr>
<td>GBytes</td>
<td>Words</td>
<td>32 bytes</td>
<td>Set Associative 2 Blocks/Set</td>
<td>( A_{31} - A_{16} )</td>
<td>2 ( A_{5} ) ( \times ) ( 12 )</td>
<td>( A_{4} - A_{2} )</td>
<td>( A_{1} )</td>
</tr>
<tr>
<td>32-bit Data address</td>
<td>32 bytes</td>
<td>2 ( A_{32} )</td>
<td>Set Associative 8 Blocks/Set</td>
<td>( A_{31} - A_{18} )</td>
<td>2 ( A_{1} ) ( \times ) ( 15 )</td>
<td>( A_{4} - A_{2} )</td>
<td>( A_{1} )</td>
</tr>
</tbody>
</table>

TAG RAM(s) and their size(s) and comparators to compare TAG(s) and their size.

In the case of Direct Mapping above, we use 1 (state a number) TAG RAM(s) of size 15-bit together with 1 (state a number) comparator(s) each of 16-bit wide.

In the case of Set Associative Mapping with 2 Blocks/Set above, we use 2 (state a number) TAG RAM(s) of size 16-bit together with 2 (state a number) comparator(s) each of 16-bit wide.

In the case of Set Associative Mapping with 8 Blocks/Set above, we use 8 (state a number) TAG RAM(s) of size 19-bit together with 8 (state a number) comparator(s) each of 18-bit wide.

The Fully Associative Mapping is prohibitively expensive because you would need 2\( 2^{7+1} = 2^{8} \) -bit wide.

In the first case of direct mapping, the main memory shall be organised in a 8-way lower-order interleaving to facilitate efficient transfer of a block of words from the main memory to cache.

The main memory organization is same (same/different) in the above 4 cache organizations, because the block size (number of words in a block) is the same.

In general, a set can potentially have a set-associativity equal to any number (not necessarily a power of 2). However, here (However, here / Here also) a set needs to be (needs to be / does not need to be) a power of 2 in size, because we started this problem with a cache size to suit DIRECT mapping which is necessarily a power of 2. Thus we changed the direct mappings to set associative mapping and ended up with power of 2 for set-associative mapping.
**Virtual Memory:**

6.1 PTBR stands for **Page Table Base Register**. It is initiated by the **operating system** (hardware / operating system) and is utilized by **MMU** (MMU / CCU) (i.e memory management unit or cache control unit) to look up **TLB** (Page Table / Cache Tag RAM).

6.2 Normally, we need to translate the virtual address (VA) put out by the processor fully to physical address (PA) before we can start indexing the cache (and finally compare the TAG. This is the case with **PIPT** (PIPT / VIPT). PIPT stands for **Physically indexed Physically Tagged**, where as VIPT stands for **Virtually Indixed and Physically Tagged**. A PIPT can be converted to a VIPT by **increasing** (increasing / decreasing / neither) the page size there by the page offset.

The following is an example of **VIPT** (PIPT / VIPT) since **page offset A13-0** have covered the **SET, WORD, and byte fields of the physical address A13-0**.