Lecture 3 Notes

Parallel Processing CS471

1 Processor Performance

1.1 The Processor Performance Equation

All computers are constructed with a clock running at a constant rate. A time event in the context of this clock is called a clock cycle or a 'tick'. Usually we refer to the clock by it's rate (e.g., 1 GHz). The CPU time for a program can be expressed as:

$$\text{CPU time} = \frac{\text{No. of CPU cycles for a program}}{\text{Clock rate}}, \tag{1}$$

or alternatively as:

$$\text{CPU time} = \text{No. of CPU clock cycles for a program} \times \text{Clock cycle time}, \tag{2}$$

where Clock cycle time $= \frac{1}{\text{Clock rate}}$.

A program is essentially a set of instructions that perform a specific task, the number of instructions for a program is called it's instruction count (IC). It's useful to know the clock cycles per instruction (CPI), which is computed as:

$$\text{CPI} = \frac{\text{No. of CPU clock cycles for a program}}{\text{Instruction count}}, \tag{3}$$

we can use this to rewrite the equation for CPU time as:

$$\text{CPU time} = \text{IC for a program} \times \text{CPI} \times \text{Clock cycle time}. \tag{4}$$

For an instruction set containing n instructions we can rewrite the formula for CPU time again:

$$\text{CPU time} = \left(\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i \right) \times \text{Clock cycle time}, \tag{5}$$

where IC_i and CPI_i are the values for an instruction i in the ISA (instruction set architecture).

We can then write the overall (program) CPI as:

$$\text{CPI} = \frac{\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i}{\text{Program IC}} = \sum_{i=1}^{n} \frac{\text{IC}_i}{\text{Program IC}} \times \text{CPI}_i \tag{6}$$

Its also useful to have an equation for the average instruction time for the purposes of comparison:

$$\text{Average instruction time} = \text{Clock cycle time} \times \text{CPI} \tag{7}$$

From Equation 4 we can see how processor performance is equally dependent on the following three characteristics: clock rate, clock cycles per instruction, and instruction count. Clock rate depends on the hardware (the processor technology) and its organization. CPI depends on the organization and the ISA. The instruction count depends on the ISA and the compiler. It can be seen how the characteristics describing processor performance are interdependent.

2 Pipeline Performance

Pipelining Recap

In Lecture 1 we discussed how a k segment pipeline should provide a k-fold speedup, and how usually we don't achieve that optimal speedup due to delays between stages (the hardware itself needs some time to advance the signal from one stage to another) and the overhead associated with pipelining itself.

Those constraints are mostly hardware related and are out of the programs control. Another set of constraints arise from dependencies within the instructions of the program itself, these are referred to as instruction-related hazards or pipeline hazards.

2.1 Pipeline Hazards

A pipeline hazard occurs occurs when the pipeline, or some portion of the pipeline, must stall (wait) because conditions do not permit continued execution. There are three types of hazards: resource/structural, data, and control.

Resource/Structural Hazards

A resource hazard occurs when two (or more) instructions that are already in the pipeline need the same resource. This causes the instructions to be executed in serial rather than in parallel.

Data Hazard

A data hazard occurs when there is a conflict in the access of an operand location. To understand a data hazard, consider two instructions to be executed sequentially, both accessing a certain register or memory location. Executed serially (one after the other) these two instructions produce a (correct) result. Executed in parallel the instructions produce a different (incorrect) result.

Data Hazard Types

There are three types of data hazards:

1. **Read After Write (RAW)** - A true dependency. An instruction modifies a memory location, and a succeeding operation reads the data in that location. The hazard occurs if the read takes place before the write operation is complete.
2. **Write After Read (WAR)** - An anti-dependancy. An instruction reads a memory location, and a succeeding operation writes to the location. The hazard occurs if the write takes place before the read operation is complete.
3. **Write After Write (WAW)** - An output dependency. Two instructions both write to the same location. The hazard occurs if the writes take place in the reverse order than that intended.

Control Hazards

Control hazards are skipped in the lecture slides.

Structural Hazard

Consider a 6-stage pipeline: Fetch Instruction (FI), Decode Instruction (DI), Calculate Operands (CO), Fetch Operands (FO), Execute Instruction (EI), Write Operand (WO).

If we try to implement this pipeline using a processor with a single memory port, this will generate a resource hazard.

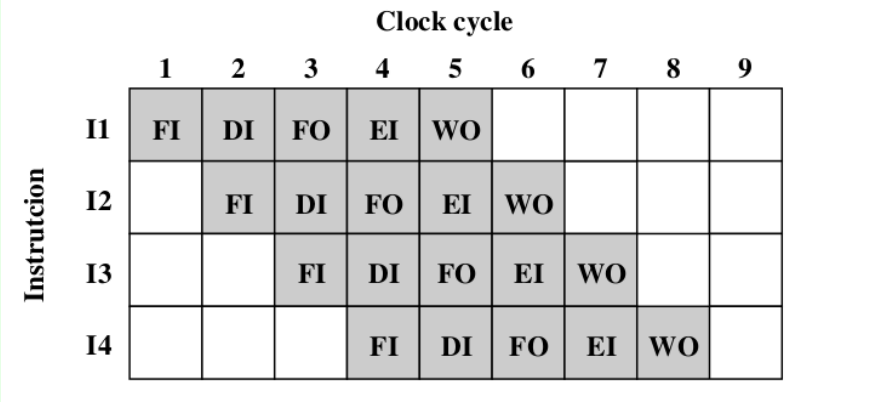

Figure 1: Space-time diagram of the proposed pipeline. The resource conflict occurs at clock cycle 3 due to the FO stage in instruction I1 and FI stage in instruction I3 both attempting to use memory for data access.

To resolve this conflict we must stall instruction I3 by one clock cycle.

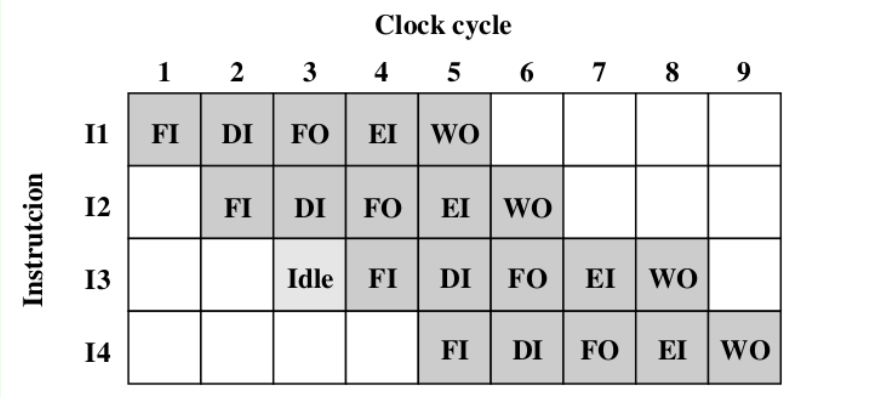

Figure 2: Space-time diagram of the resolved pipeline.

This example demonstrates a resource hazard involving memory. However, this conflict can occur with any limited resource in a processor.

Data Hazard (RAW)

Consider the following x86 machine instruction sequence:

```
ADD EAX, EBX
SUB ECX, EAX
```

The first instruction adds the content of the registers `EAX` and `EBX` and stores them in `EAX`. The second instruction subtracts the content of the register `EAX` from `ECX` and stores them in `ECX`.

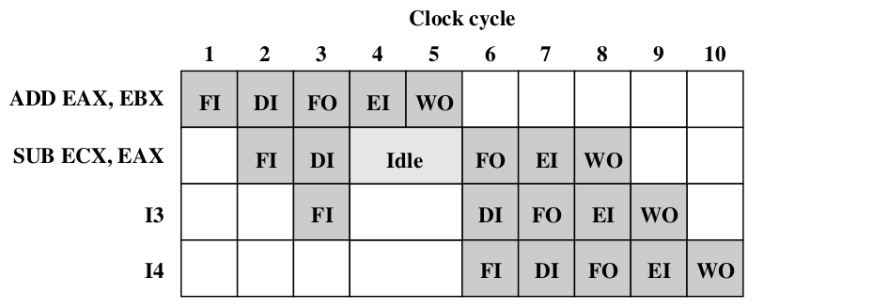

Figure 3: Space-time diagram of the proposed pipeline. In order to ensure correct operation, we need to stall the pipeline at clock cycles 4 and 5, so that the first instruction's write operation is complete, before the second instruction reads the data in that register.

Data dependencies like these are detected during the decoding stage (DI), usually by the compiler, which compares between destination registers of earlier instructions and the registers needed by the current instruction.

2.2 Forwarding/Bypassing

Forwarding is a technique using special hardware to detect and resolve data hazards. Forwarding allows an instruction to use the result of a previous instruction before it has been written back to the register file. This is done by providing a path for the data to be "forwarded" from one pipeline stage directly to another. This prevents the need for stalling (in certain cases).

Forwarding requires additional hardware paths to route data from one pipeline stage to an earlier one. These paths are usually EX-to-EX; forwarding data from the output of the EX stage to the input of the EX stage of the following instruction, or MEM-to-EX; forwarding data from the output of the MEM stage to the EX stage input.

Pipeline Registers

It's useful to think of the registers between each pipeline stage. The pipeline stages use these registers to communicate.

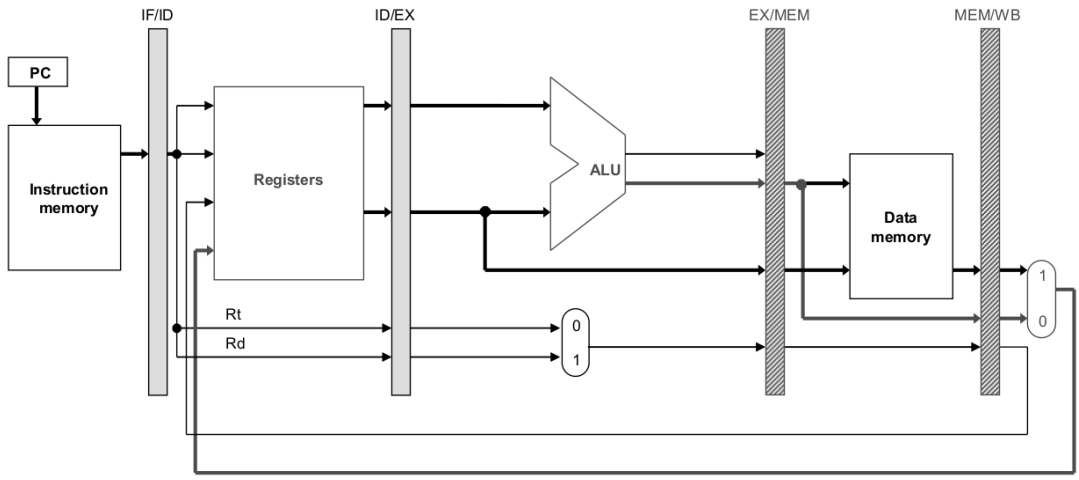

Figure 4: Circuit diagram of a 5-stage pipeline with registers between each stage.

Using these registers we can define if conditions for forwarding:

- **Condition 1 (EX Hazard)** - **IF** the destination register of an instruction in the EX/MEM pipeline register matches a source register of the instruction in the ID/EX pipeline register, **AND** the instruction in the EX/MEM stage is goint to write to a register (RegWrite is true),

  THEN forward data from the EX/MEM pipeline register to the ALU input.

- **Condition 2 (MEM Hazard)** - **IF** the destination register of an instruction in the MEM/WB pipeline register matches a source register of the instruction in the ID/EX pipeline register, **AND** the instruction in the MEM/WB stage is going to write to a register (RegWrite is true),

  THEN forward data from the MEM/WB pipeline register to the ALU input.

Note 1. In condition 2 we also usually assert that the source register is not the destination register, i.e., the instruction isn't writing back to it's source register.

The above if conditions are evaluated in a component called the forwarding unit. Based on the output from the forwarding conditions we use a multiplexer to select the correct data to forward.

Forwarding Connections

In the forwarding example the professor uses in the slides (*see slides 42-45*), based on the way forwarding is demonstrated, he seems to use a pipeline with only MEM-EX connections for forwarding. Usually when considering a pipeline with forwarding capabilities we assume it to have both EX-EX and MEM-EX connections (like the one shown in the circuit diagram *see slide 33*), unless explicitly specified. See examples below for more clarification.

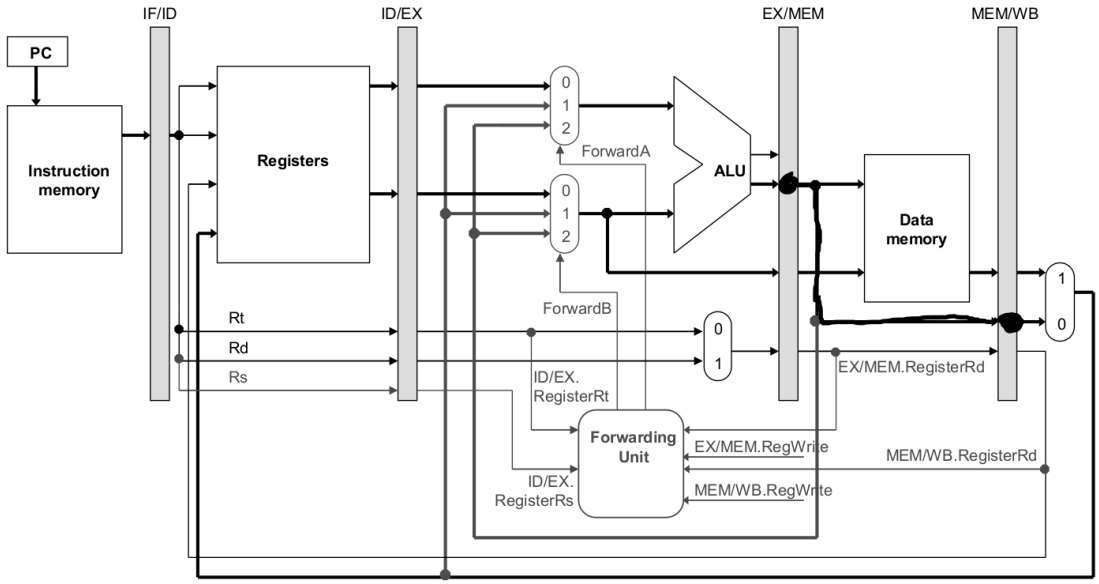

Figure 5: Circuit diagram of a 5-stage pipeline with forwarding capabilities. "RegWrite" inputs are received from the control unit (CU)

Forwarding Using Space Time Diagram #1

Consider the following instructions to be pipelined using a 5-stage pipeline:

```
LD R1, 0(R2)
SUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9
```

Assuming that each of the instructions takes a single clock cycle in each pipeline stage, and that forwarding hardware exists between the MEM and EX stages in the pipeline, we can determine the number of clock cycles needed for execution using a space-time diagram:

Instruction	**Cycle 1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
LD	IF	ID	EX	MEM	WB				
SUB		IF	ID	Stall	EX	MEM	WB		
AND			IF	Stall	ID	EX	MEM	WB	
OR				IF	Stall	ID	EX	MEM	WB

Figure 6: The `LD` instruction executes normally without need for forwarding. The `SUB` instruction uses the operand `R1` which is being modified by the previous `LD` instruction, so it must stall till that instruction completes it's MEM stage, at which point it can be forwarded (forwarding takes place from `LD` instruction in MEM stage at clock cycle 4 to `SUB` instruction in EX stage at clock cycle 5). The `AND` instruction stalls in cycle 3 since the previous `SUB` instruction is still in the ID stage so it cannot proceed there. Similarly the `OR` instruction stalls in clock cycle 5 since the previous `AND` instruction is occupying the ID stage.

Forwarding Using Space Time Diagram #2

Consider the following instructions to be pipelined using a 5-stage pipeline:

```
MUL R5, R0, R1
DIV R6, R2, R3
ADD R7, R5, R6
SUB R8, R7, R4
```

The IF,ID, MEM and WB stages take 1 clock cycle each for any instruction. The EX stage takes 1 clock cycle for ADD and SUB instruction, 3 clock cycles for MUL instruction and 5 clock cycles for DIV instruction. The pipelined processor uses operand forwarding from the MEM stage to the EX stage.

To determine how many clock cycles it would take for the execution of these instructions, we draw the space-time diagram:

Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
`MUL`	IF	ID	EX_1	EX_2	EX_3	MEM	WB							
`DIV`		IF	ID	EX_1	EX_2	EX_3	EX_4	EX_5	MEM	WB				
`ADD`			IF	ID	Stall	Stall	Stall	Stall	Stall	EX	MEM	WB		
`SUB`				IF	ID	Stall	Stall	Stall	Stall	Stall	Stall	EX	MEM	WB

Figure 7: The `MUL` and `DIV` instructions execute normally without the need for forwarding. The `ADD` instruction uses the operand `R6` which is being modified by the previous `DIV` instruction, so it must stall till that instruction completes it's MEM stage, at which point it can be forwarded (forwarding takes place from `DIV` instruction in MEM stage at clock cycle 9 to `ADD` instruction in EX stage at clock cycle 10). Similarly the `SUB` instruction uses the operand `R7` from the previous `ADD` instruction, so it stalls till it that operand can be forwarded (forwarding takes place from `ADD` instruction in MEM stage at clock cycle 11 to `ADD` instruction in EX stage at clock cycle 12).

Using the space-time diagram we can determine that the pipelined execution of these instruction will take 14 clock cycles, completing at clock cycle 15.

You'll notice that despite using forwarding, the pipeline still has to stall. Stalling is not always avoidable, but in this case we avoid having to stall for an extra clock cycle in the `ADD` and `SUB` operations, which would have occurred without forwarding.

Comparison With Unpipelined Performance

Unpipelined:

- `MUL` instruction would take 7 clock cycles.
- `DIV` instruction would take 10 clock cycles.
- `ADD` and `SUB` instructions would each take 5 clock cycles.

Making for a total of 26 clock cycles.

The speedup here being: $S_{\text{actual}} = \frac{26}{15} \approx 1.733$. We can compare this to the ideal speedup:

$$S_{\text{ideal}} = \frac{nt_n}{(k+n-1)t_p} = \frac{26}{(5+4-1)1} = 3.25$$

Forwarding Using Circuit Diagram

Consider the following instructions to be pipelined:

```
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```

Assume that our pipeline is set up using the circuit in figure 5, and that each register contains its register number +100: `$1` contains 101, `$2` contains 102, and so on.

To determine the signals being processed at each component we 'dry run' the circuit with our instructions and determine the signal values at each clock cycle

The first two cycles can be skipped. All that occurs is that the first two instructions go through the IF and ID phases, and the values of their operand are loaded from the registers.

Clock Cycle 3

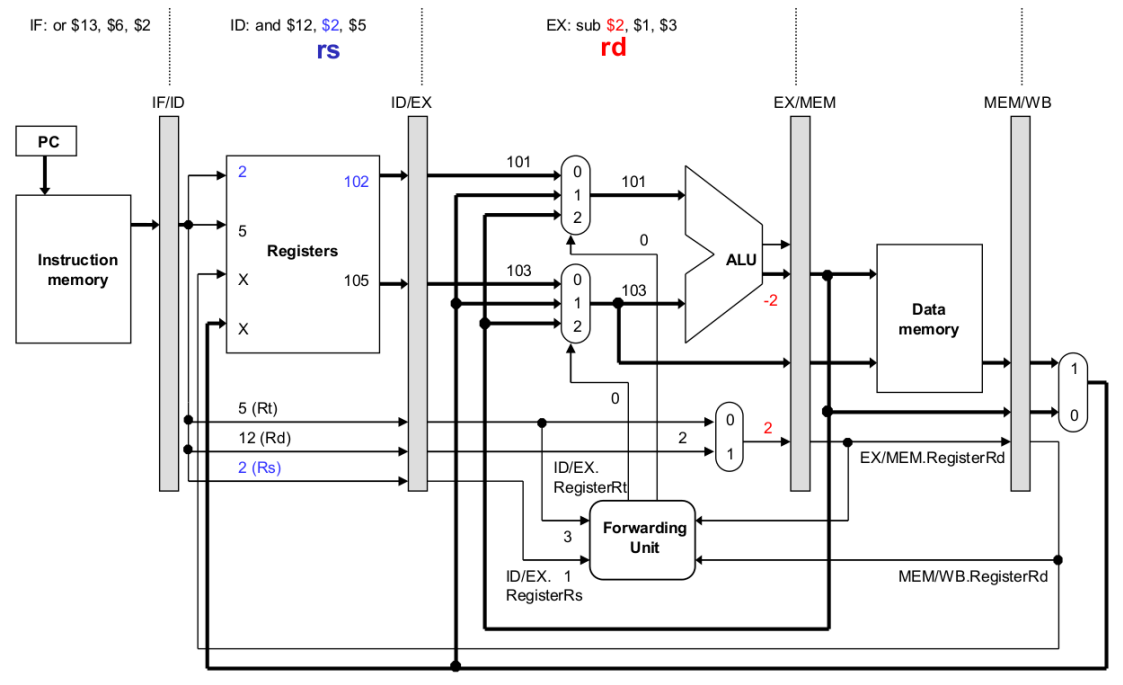

Figure 8: Pipeline circuit at clock cycle 3.

For `sub $2, $1, $3` which is in the EX stage: the ALU unit performs the subtraction of `103` (value in `$3`) from `101` (value in `$1`) resulting in `-2`.

To determine if forwarding is needed the forwarding unit evaluates 2 values: `FowardA` and `ForwardB`, which indicate whether the ALU inputs from the next instruction will need the value from the current instruction.

Its useful to think of the forwarding unit as a sort of function, we can then define it as follows:

```
        # Inputs:
        #   EX_MEM_RegWrite: Boolean indicating if EX/MEM stage instruction writes to a register
        #   MEM_WB_RegWrite: Boolean indicating if MEM/WB stage instruction writes to a register
        #   EX_MEM_RegisterRd: Destination register number of EX/MEM stage instruction
        #   MEM_WB_RegisterRd: Destination register number of MEM/WB stage instruction
        #   ID_EX_RegisterRs: Source register Rs of ID/EX stage instruction
        #   ID_EX_RegisterRt: Source register Rt of ID/EX stage instruction

        # Outputs:
        #   ForwardA: 2-bit control signal for ALU input A multiplexer
        #   ForwardB: 2-bit control signal for ALU input B multiplexer

        def ForwardingUnit(
            EX_MEM_RegWrite, MEM_WB_RegWrite,
            EX_MEM_RegisterRd, MEM_WB_RegisterRd,
            ID_EX_RegisterRs, ID_EX_RegisterRt
        ):
            # Initialize forwarding controls to default (no forwarding)
            ForwardA = "00"
            ForwardB = "00"

            # Forwarding logic for source operand A (RegisterRs)
            if EX_MEM_RegWrite and (EX_MEM_RegisterRd != 0) and (EX_MEM_RegisterRd ==
            ↪  ID_EX_RegisterRs):
                # EX hazard: Forward from EX/MEM pipeline stage
                ForwardA = "10"
            elif MEM_WB_RegWrite and (MEM_WB_RegisterRd != 0) and (MEM_WB_RegisterRd ==
            ↪  ID_EX_RegisterRs):
                # MEM hazard: Forward from MEM/WB pipeline stage
                ForwardA = "01"

            # Forwarding logic for source operand B (RegisterRt)
            if EX_MEM_RegWrite and (EX_MEM_RegisterRd != 0) and (EX_MEM_RegisterRd ==
            ↪  ID_EX_RegisterRt):
                # EX hazard: Forward from EX/MEM pipeline stage
                ForwardB = "10"
            elif MEM_WB_RegWrite and (MEM_WB_RegisterRd != 0) and (MEM_WB_RegisterRd ==
            ↪  ID_EX_RegisterRt):
                # MEM hazard: Forward from MEM/WB pipeline stage
                ForwardB = "01"

            # Return the forwarding control signals
            return ForwardA, ForwardB
```

At the current cycle the outputs of the forwarding unit are still the default (zeros) for `ForwardA` and `ForwardA`. This is because the output value from the ALU needs to first be placed in the EX/MEM pipeline register before it is forwarded, which will happen in the next clock cycle.

Clock Cycle 4

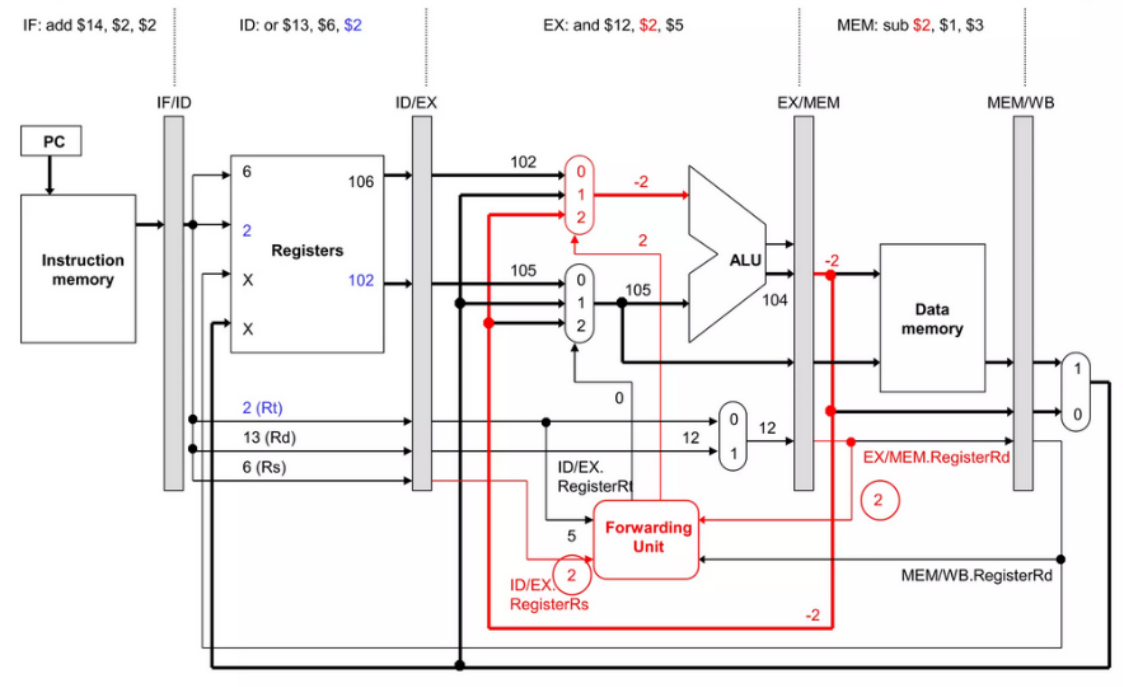

Figure 9: Pipeline circuit at clock cycle 4.

For `sub $2, $1, $3` which is now in the MEM stage: The result `-2` from the `sub` instruction is now held in the EX/MEM pipeline register, ready to be forwarded.

For `and $12, $2, $5` which is now in the EX stage: the `and` operation in this instruction needs the value of `$2` from the previous instruction, which is in the EX/MEM pipeline register. We can imagine the forwarding unit function being called as follows:

```
FowardA, ForwardB = ForwardingUnit(
    EX_MEM_RegWrite=1, MEM_WB_RegWrite=0,
    EX_MEM_RegisterRd=2, MEM_WB_RegisterRd=0,
    ID_EX_RegisterRs=2, ID_EX_RegisterRt=3
)
```

Based on our code, the following inputs would trigger the 'EX Hazard' condition for source operand A, this produces: `ForwardA = 01 (2)` and `ForwardB = 00 (0)`.

Moving on to the multiplexers. The multiplexers act as a selector, that selects the correct values for the ALU operands based on the control signals (`ForwardA` and `ForwardB`) from the forwarding unit

For completeness we can also represent the multiplexers as functions.

```
# Inputs:
#   ID_EX.ReadDataA: Operand A (rs) from the register file (ID/EX pipeline register)
#   ID_EX.ReadDataB: Operand B (rt) from the register file (ID/EX pipeline register)
#   EX_MEM.ALUResult: ALU result from the EX/MEM pipeline stage
#   MEM_WB.WriteData: Data from the MEM/WB pipeline stage

# Control signals from the forwarding unit:
#   ForwardA: 2-bit control signal for ALU input A
#   ForwardB: 2-bit control signal for ALU input B

# Outputs:
#   ALU_input_A: First operand for the ALU
#   ALU_input_B: Second operand for the ALU

def MUX_ALU_A(ID_EX.ReadDataA, EX_MEM.ALUResult, MEM_WB.WriteData, ForwardA):
    elif ForwardA == "10":
        ALU_input_A = EX_MEM.ALUResult # EX Hazard
    elif ForwardA == "01":
        ALU_input_A = MEM_WB.WriteData # MEM Hazard
    else: # 00
        ALU_input_A = ID_EX.ReadDataA  # Default case

def MUX_ALU_B(ID_EX.ReadDataB, EX_MEM.ALUResult, MEM_WB.WriteData, ForwardB):
    elif ForwardB == "10":
        ALU_input_B = EX_MEM.ALUResult # EX Hazard
    elif ForwardA == "01":
        ALU_input_B = MEM_WB.WriteData # MEM Hazard
    else: # 00
        ALU_input_B = ID_EX.ReadDataB  # Default case
```

In the current cycle since we have an EX hazard in operand B, the multiplexer selects the ALU result from the previous instruction (`EX_MEM.ALUResult`), which contains the desired value of `$2`.

Clock Cycle 5

For `sub $2, $1, $3` which is now in the WB stage: the result of `-2` is written into register `$2`, completing the instruction.

For `and $12, $2, $5` which is now in the MEM stage: the `and` instruction's result (`2 & 105`) is stored in the MEM/WB pipeline register, preparing for a write-back to `$12`.

For `or $13, $6, $2` which is now in the EX stage: the `or` instruction needs the value of register `$2` which we can get from the MEM/WB pipeline register. The forwarding unit will detect this:

```
FowardA, ForwardB = ForwardingUnit(
    EX_MEM_RegWrite=1, MEM_WB_RegWrite=1,
    EX_MEM_RegisterRd=12, MEM_WB_RegisterRd=2,
    ID_EX_RegisterRs=6, ID_EX_RegisterRt=2)
```

These inputs trigger the 'MEM Hazard' condition for source operand B. this produces: `ForwardA = 00 (0)` and `ForwardB = 01 (1)`.

The B operand multiplexer would then select the data from the MEM/WB pipeline register for the input (`MEM_WB.WriteData`), which contains the desired value of `$2`.

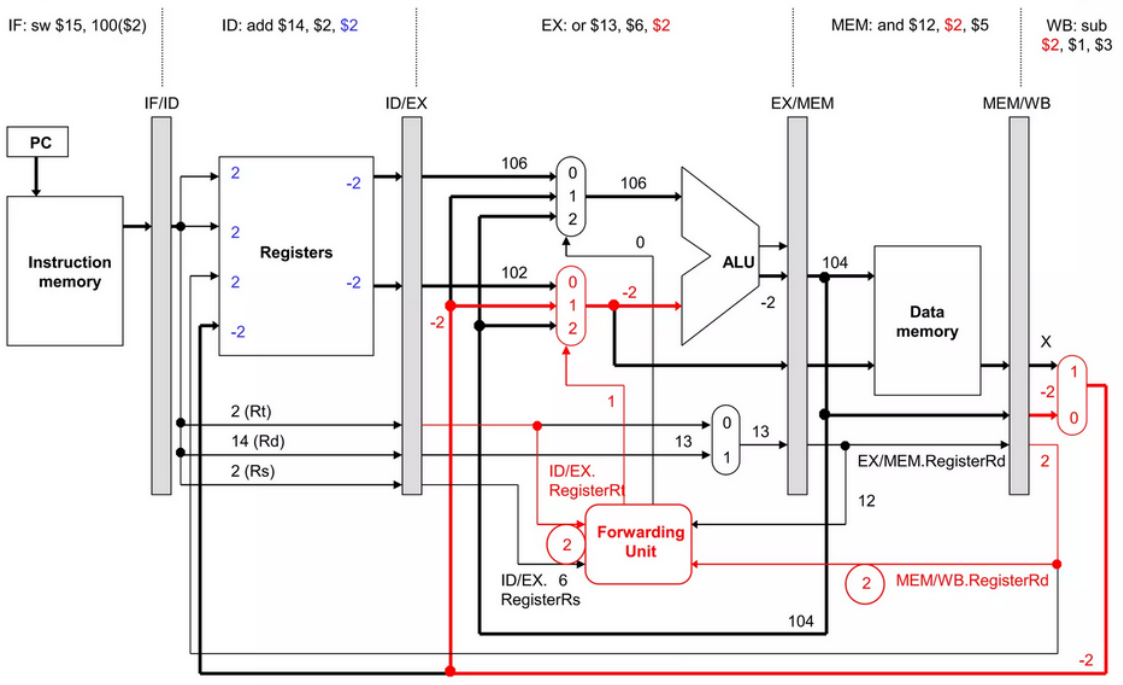

Figure 10: Pipeline circuit at clock cycle 5.

Space Time Diagram

All the signals and "functions" being performed might make the process seem more complicated than it is. It can be helpful to look at the space-time diagram to understand what is happening better:

Instruction	Cycle 1	2	3	4	5	6	7	8	9
sub $2,$1,$3	IF	ID	EX	MEM	WB				
and $12,$2,$5		IF	ID	EX	MEM	WB			
or $13,$6,$2			IF	ID	EX	MEM	WB		
add $14,$2,$2				IF	ID	EX	MEM	WB	
sw $15,100($2)					IF	ID	EX	MEM	WB

Figure 11: Forwarding allows for the pipeline to execute with no stalls. Forwarding occurs as follows: - `$2` is forwarded from the EX stage of the `sub` instruction in clock cycle 3 to the EX stage of the `and` instruction in clock cycle 4 (EX-EX forwarding) - `$2` is forwarded from the MEM stage of the `sub` instruction in clock cycle 4 to the EX stage of the `or` instruction in clock cycle 5 (MEM-EX forwarding)

References