1. Instruction fetch (IF): The instruction is fetched from memory and placed in the instruction register (IR).

2. Instruction decode (ID): The bits of the instruction are decoded into control signals. Operands are moved from registers or immediate fields to working registers. For branch instructions, the branch condition is tested and the branch address computed.

3. Execution (EX): The instruction is executed. Specifically, if the instruction is an arithmetic or logical operation, its results are computed. If it is a load-store instruction, the address is computed. All this is done by an elaborate logic circuit called the arithmetic-logical unit (ALU).

4. Memory read/write (ME): If the instruction is a load-store, the memory is read or written.

5. Write back (WB): The results of the operation are written to the destination register.

• • •

cycle	IF	ID	EX	ME	WB
1	I1				
2	I2	I1			
3	I3	I2	I1		
4	I4	I3	I2	I1	
5	I5	I4	I3	I2	I1
6	I6	I5	I4	I3	I2
.	.	.	.	.	.

In

cycle	IF	ID	EX	ME	WB
1	I1				
2	I2	I1			
3	I3	I2	I1		
4	I4	I3	I2	I1	
5	I5	I4	I3	I2	I1
6	I6	I5	I4	I3	I2
.	.	.	.	.	.

suppose I2 is a load. Then we have the following execution sequence

cycle	IF	ID	EX	ME	WB
1	I1				
2	LD	I1			
3	I3	LD	I1		
4	I4	I3	LD	I1	
5	–	I4	I3	LD	I1
6	I5	–	I4	I3	LD
6	I6	I5	–	I4	I3
.	.	.	.	.	.

Consider the following sequence of instructions.

```
DADD   R1, R2, R3
DSUB   R1, #10
```

The execution diagram is

cycle	IF	ID	EX	ME	WB
1	AD				
2	SB	AD			
3	XX	SB	AD		
4	XX	SB	–	AD	
5	XX	SB	–	–	AD
6	XX	SB	–	–	–
7	XX	XX	SB	–	–
.	.	.	.	.	.

At cycle 4, SB cannot complete the ID stage, because the new value is not in R1.

A cure is for the CPU to move the new value of R1 into the the ID/EX register as AD completes the EX stage. This process is called **forwarding.**

ANOTHER DATA HAZARD

Consider the following instructions.

```
LD      R1, 0(R2)
DSUB    R1, #10
```

The execution sequence is

cycle	IF	ID	EX	ME	WB
1	LD				
2	SB	LD			
3	XX	SB	LD		
4	XX	SB	–	LD	
5	XX	XX	SB	–	LD
6	XX	XX	XX	SB	–
.	.	.	.	.	.

Forwarding can take place only as SD completes the ME stage. This leaves a stall in the pipeline.

One cure would be to find another instruction AA to execute between LD and SB.

If the instruction does not depend on R1, we get the following sequence

cycle	IF	ID	EX	ME	WB
1	LD				
2	AA	LD			
3	SB	AA	LD		
4	XX	SB	AA	LD	
5	XX	XX	SB	AA	LD
6	XX	XX	XX	SB	AA
.	.	.	.	.	.

The compiler must find such an instruction. The process is called **rescheduling.**

Consider a branch instruction BR. The address that it jumps to, called the target, is only known at the end of the ID stage. Hence if we denote the target instruction by TG, we have the following execution sequence.

cycle	IF	ID	EX	ME	WB
1	BR				
2	–	BR			
3	TG	–	BR		
4	XX	TG	–	BR	
5	XX	XX	TG	–	BR
6	XX	XX	XX	TG	–
.	.	.	.	.	.

The cure is to schedule an instruction to fill the stall.

• • •

In **static scheduling** the compiler chooses the instruction. There are three possibilities.

1. The compiler finds an instruction that is independent of the branch.
2. The compiler assumes the branch will be taken and execute the target.
3. The compiler assumes the branch will not be taken and execute the target.

In loops assuming the target is the first instruction in the loop is a good strategy.

• • •

In **dynamic scheduling** the CPU chooses the instruction.

Assume

1. Floating-point adds can be pipelined with a latency of three cycles.
2. All other operations require one cycle.

• • •

Here is the AXPY code.

```
1.  Loop: L.D    F1, 0(R1)  ; 1
2.        L.D    F2, 0(R2)  ; 1
3.        MUL.D  F1, F0, F1 ; 3
4.        ADD.D  F2, F2, F1 ; 3
5.        S.D    0(R2), F2  ; 1
6.        DADDI  R1, #8     ; 1
7.        DADDI  R2, #8     ; 1
8.        DSUB   R4, R1, R3 ; 1
9.        BNZE   R4, Loop   ; 1
```

(1)

The total number of cycles is 13. No savings.

Assuming `n` is divisible by 3, we unroll the loop as follows.

```
for i = 1:3:n
   y(i) = y(i) + a*x(i);
   y(i+1) = y(i+1) + a*x(i+1);
   y(i+2) = y(i+2) + a*x(i+2);
end
```

The unrolled AXPY code is

```
 1. Loop: L.D     F1,  0(R1)
 2.       L.D     F2,  8(R1)
 3.       L.D     F3,  16(R1)
 4.       L.D     F4,  0(R2)
 5.       L.D     F5, 8(R2)
 6.       L.D     F6, 16(R2)
 7.       MUL.D   F1, F0, F1
 8.       MUL.D   F2, F0, F2
 9.       MUL.D   F3, F0, F3
10.       ADD.D   F4, F1, F4
11.       ADD.D   F5, F2, F5
12.       ADD.D   F6, F3, F6
13.       S.D     0(R2), F4
14.       S.D     8(R2), F5
15.       S.D     16(R2), F6
16.       DADDI   R1, #24
17.       DADDI   R2, #24
18.       DSUB    R4, R1, R3
19.       BNZE    R4, Loop
```

It requires

$$6\,\texttt{L.D} + 3\,\texttt{MUL.D} + 3\,\texttt{ADD.D} + 3\,\texttt{S.D} + 4\,\texttt{loop} = 19\,\text{cycles.}$$

for an effective rate of $19/3 = 6\frac{1}{3}$ cycles.

When $k = 3$ we have

$$6\,\texttt{L.D} + 3\,\texttt{MUL.D} + 3\,\texttt{ADD.D} + 3\,\texttt{S.D} + 4\,\texttt{loop} = 19\,\text{cycles}.$$

For an average 6.3 cycles

• • •

What $k \geq 3$ we have

$$2k\,\texttt{L.D} + k\,\texttt{MUL.D} + k\,\texttt{ADD.D} + k\,\texttt{S.D} + 4\,\texttt{loop} = 5k + 4\,\text{cycles}$$

so that the effective rate is $5 + 4/k$. For $k = 5$ this is 5.8.

Increasing k increases the number of registers.

• • •

When $k \leq 3$ we have

$$2k\,\texttt{L.D} + 3\,\texttt{MUL.D} + 3\,\texttt{ADD.D} + k\,\texttt{S.D} + 4\,\texttt{loop} = 3k + 10\,\text{cycles}.$$

The effective rate is $3 + 10/k$.

$$k = 1,\ 13; \quad k = 2,\ 8; \quad k = 3,\ 6.3;$$

The good decrease is due to the fact we are decreasing both average loop overhead and using the pipeline more efficiently

When n is not divisible by 3.

```
m = rem(n, 3);
for i=1:m
    y(i) = y(i) + a*x(i);
end
for i = m+1:3:n
   y(i) = y(i) + a*x(i);
   y(i+1) = y(i+1) + a*x(i+1);
   y(i+2) = y(i+2) + a*x(i+2);
end
```

```
dot = 0;
for i=1:n
   dot = dot + x(i)*y(i);
end
```

• • •

```
 1.        L.D    F0, zero
 2.  Loop: L.D    F1, 0(R1)  ; load x(i)
 3.        L.D    F2, 0(R2)  ; load y(i)
 4.        MUL.D  F1, F1, F2 ; x(i)*y(i)
 5.        ADD.D  F0, F0, F1 ; F0 + x(i)*y(i)
 6.        DADDI  R1, #8     ; update address of x(i)
 7.        DADDI  R2, #8     ; update address of y(i)
 8.        DSUB   R4, R1, R3 ; zero at end of loop
 9.        BNZE   R4, Loop
10.        S.D    dot, F0    ; store F0 in dot
```

• • •

```
s1 = 0; s2 = 0; s3 = 0;
for i=1:3:n
   s1 = s1 + x(i)*y(i);
   s2 = s2 + x(i+1)*y(i+1);
   s3 = s3 + x(i+2)*y(i+2);
end
dot = s1 + s2 + s3;
```