WAR Dependencies

```
i1: mul r1, r2, r3
i2: add r2, r4, r3
```

WAR dependencies are false dependencies since they can be eliminated by register renaming; that is, the destination register of the affected instruction should be renamed to a register name that has not yet been used.

```
i1: mul r1, r2, r3
i2: add r6, r4, r5
```

[r2 in i2 has been renamed to r6]

WAW dependencies

```
i1: mul r1, r2, r3
i2: add r1, r4, r5
```

Data dependencies in loops:

Instructions belonging to a particular iteration may be dependent on instruction belonging to previous loop iterations. This kind of data dependency is called recurrence or inter-iteration data-dependency or loop-carried data-dependency.

```
do I=2,n
        X(I)=AX(I-1)+B
enddo
```

Control Dependencies

```
        mul r1, r2, r3
        j3  zproc
        sub r4, r1, r1
        .
        .
        .
zproc:  load r1, x
```

The actual path of execution depends on the outcome of the multiplication.

All conditional control instructions such as conditional branches, calls, and so on, impose dependencies on the logically subsequent instructions. These dependencies are referred to as control dependencies.

Resource Dependencies:

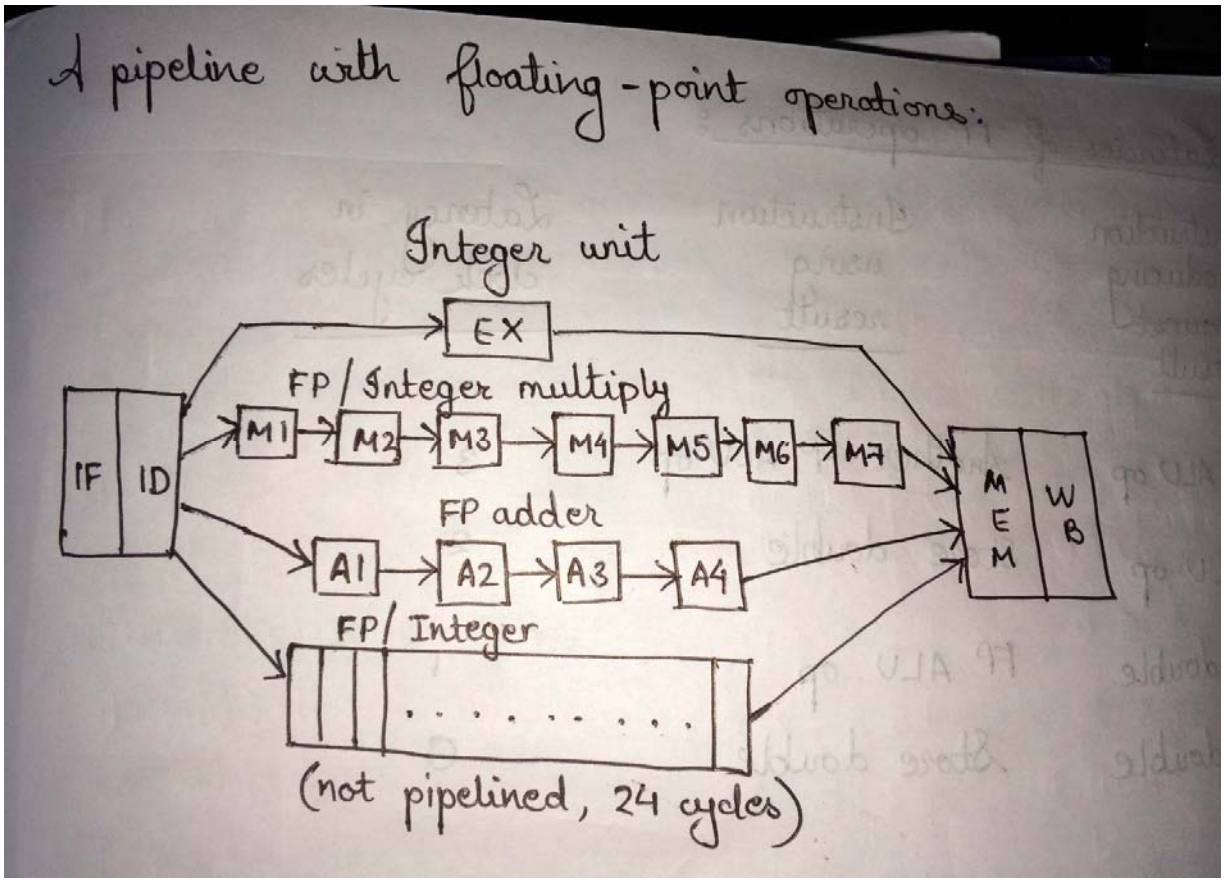
An instruction is resource-dependent on a previously issued instruction if it requires requests a hardware resource which is still being used by a previously issued instruction.

If, for instance, only a single non-pipelined division unit is available, as is usual in ILP-processors, then in the code sequence

```
div r1, r2, r3
div r4, r2, r5
```

The second division instruction is resource-dependent on the first one and cannot be executed in parallel if there is only a single division unit available.

A pipeline with floating-point operations



Latencies and Initiation Intervals

Functional Unit	Latency	Initiation Interval
Integer ALU	0	1
Data memory (integer and FP loads)	1	1
FP Add	3	1
FP Multiply (also integer multiplication)	6	1
FP Divide	24	25

Latency= The number of intervening cycles between an instruction that produces a result and an instruction that uses the result

Initiation interval= Number of cycles that must elapse between issuing 2 operations of the same type

Latencies of FP Operations:

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store Double	2
Load Double	FP ALU op	1
Load Double	Store Double	0

```
for ( i=1000 ; i>0 ; i=i-1)
        x[i]=x[i]+s;
```

Straight-forward Translation:

```
Loop:   L.D F0, O(R1)          ;       F0=array element
        ADD.D F4, F0, F2       ;       add scalar in F2
        S.D F4, O(R1)          ;       store result
        DADDUI R1, R1, #-8     ;       decrement pointer
                               ;       8 bytes
        BNE R1, R2, Loop       ;       branch R1 != R2
```

Without any scheduling, the loop will execute as follows:

Label	Opcode	Operands	Clock Cycle Issued *
Loop:	L.D	F0, O(R1)	1
	stall	----------	2
	ADD.D	F4, F0, F2	3
	stall	----------	4
	stall	----------	5
	S.D	F4, O(R1)	6
	DADDUI	R1, R1, #-8	7
	stall	----------	8
	BNE	R1, R2, Loop	9
	stall	----------	10

**10 cycles per iteration, 5 stalls*

With Scheduling *(6 cycles per iteration. Only 1 stall):*

```
Loop:   L.D      F0, O(R1)
        DADDUI   R1, R1, #-8
        ADD.D    F4, F0, F2
        stall
        BNE      R1, R2, Loop     ; Delayed brand
        S.D      F4, 8(R1)        ; Altered and interchanged with DADDUI
```

Loop Unrolling:

A simple scheme for increasing the number of instructions relative to the branch and overhead instructions is *loop unrolling*. Unrolling simply replicates the loop body multiple times, adjusting the loop termination code.

```
Loop:  L.D     F0, O(R1)
                                  < ---------- stall
       ADD.D   F4, F0, F2
                                  < ---------- 2 x stall
1      S.D     F4, O(R1)          ; drop DADDUI & BNE
-------------------------------------------------------
       L.D     F6, -8(R1)
                                  < ---------- stall
       ADD.D   F8, F6, F2
                                  < ---------- 2 x stall
2      S.D     F8, -8(R1)         ; drop DADDUI & BNE
-------------------------------------------------------
       L.D     F10, -16(R1)
                                  < ---------- stall
       ADD.D   F12, F10, F2
                                  < ---------- 2 x stall
3      S.D     F12, -16(R1)       ; drop DADDUI & BNE
-------------------------------------------------------
       L.D     F14, -24(R1)
                                  < ---------- stall
       ADD.D   F16, F14, F2
                                  < ---------- 2 x stall
4      S.D     F16, -24(R1)
-------------------------------------------------------
       DADDUI  R1, R1, #-32
                                  < ---------- stall
       BNE     R1, R2, Loop
                                  < ---------- stall
```

N.B. *(R2) + 32= STARTING ADDRESS OF LAST 4 ELEMENTS*
NO. OF LOOP ITERATIONS IS ASSUMED TO BE A MULTIPLE OF 4, I.E., R1 IS INITIALLY A MULTIPLE OF 32

Unrolled Loop: *14 issue cycles + 14 stalls*
= > 28 cycles per iteration
= > 7 cycles per element

Scheduling the UNROLLED Loop:

```
L.D       F0, 0(R1)
L.D       F6, -8(R1)
L.D       F10, -16(R1)
L.D       F14, -24(R1)
ADD.D     F4, F0, F2
ADD.D     F8, F6, F2
ADD.D     F12, F10, F2
ADD.D     F16, F14, F2
S.D       F4, 0(R1)
S.D       F8, -8(R1)
DADDUI R1, R1, #-32
S.D       F12, 16(R1)
BNE R1, R2, Loop
S.D       F16, 8(R1)        ;          8-32 = -24
```

Execution Time = 14 cycles per iteration

i.e. 14/4 = 3.5 cycles per element of the array (compared to 6 cycles per element)

Dynamic Scheduling:

MIPS Floating point unit:

Dynamic Scheduling:

MIPS Floating Point Unit: