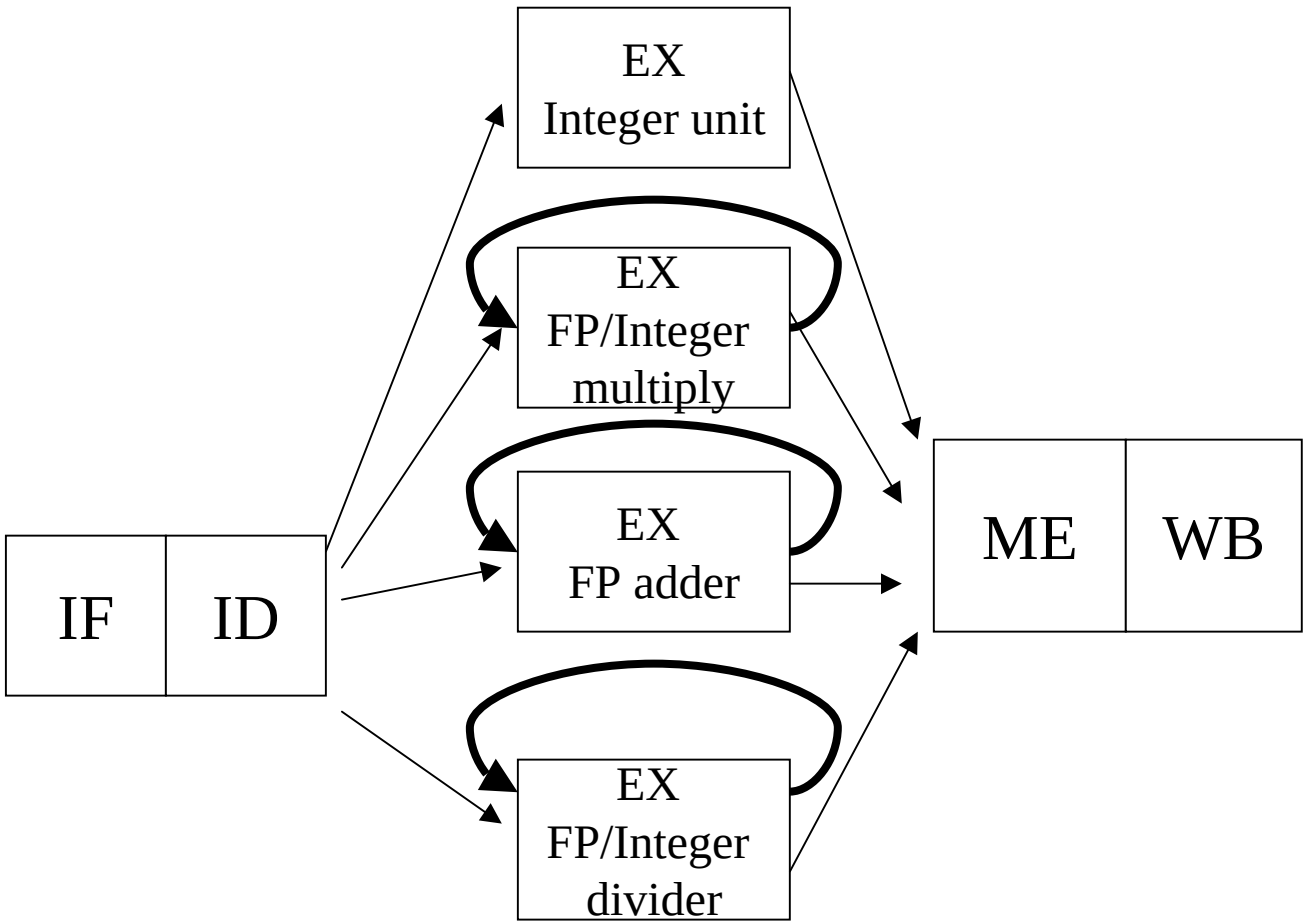
Lets now look at floating point operations

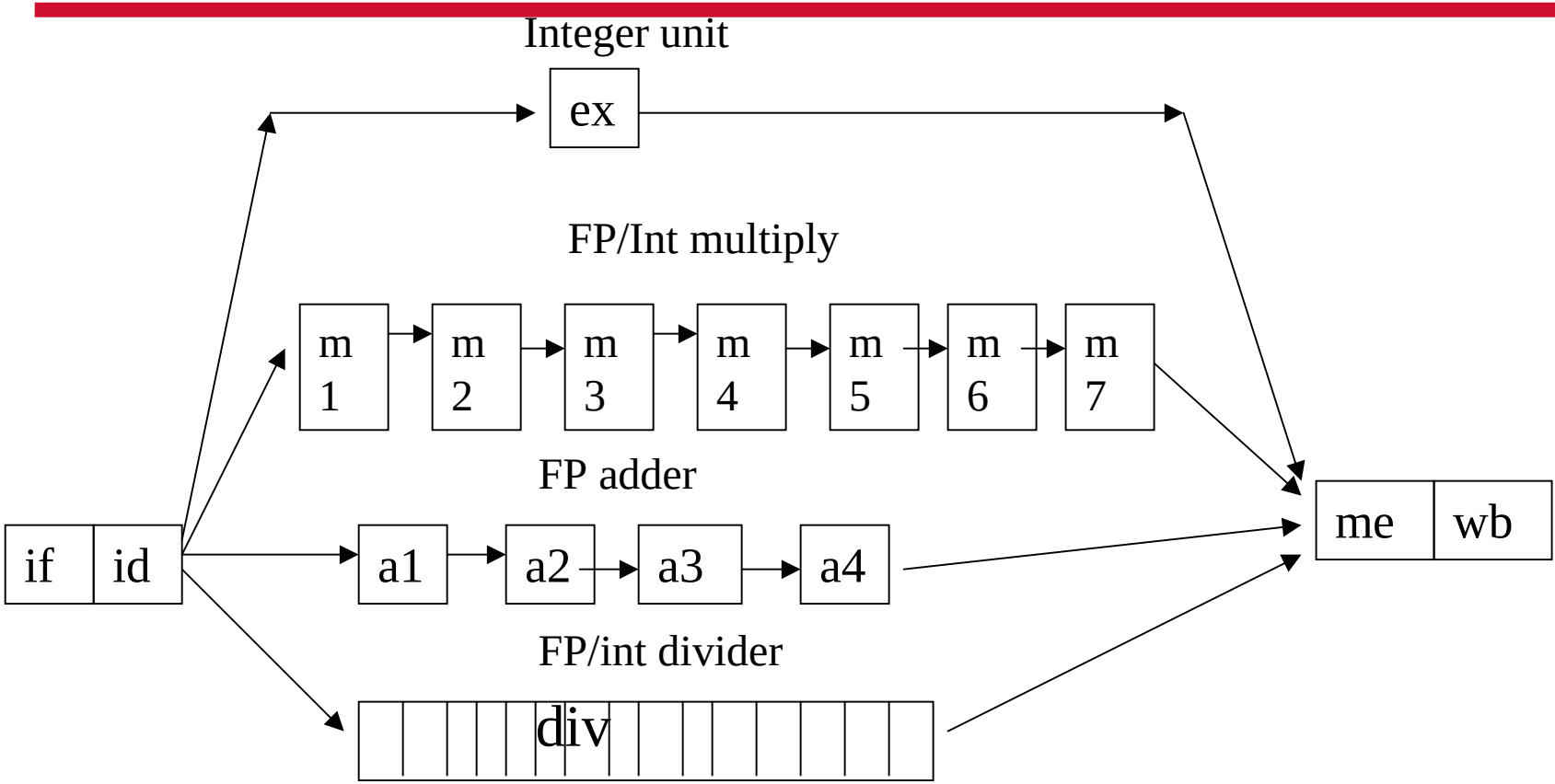
- Impractical to require all FP ops to complete in one or even two clock cycles since that would make slower clock or using enormous logic in FP units
- FP pipeline will allow for a latency for operations
- In FP pipelines
  - The EX cycle can be repeated many times and the number of repetitions vary for different operations
  - There may be multiple FP functional units

- Lets build with four separate functional units
  - Main integer unit handles loads, stores, integer ALU ops, and branches
  - FP and integer multiplier
  - FP adder that handles FP add, subtract, and conversion
  - FP and integer divider
- These execution stages of these functional units are not pipelined

Resulting structure of previous machine with 3 additional unpipelined FP functional units



Lets build pipelined FP functional units out of the last one


Integer unit
ex
FP/Int multiply
m 1
m 2
m 3
m 4
m 5
m 6
m 7
FP adder
if
id
a1
a2
a3
a4
me
wb
FP/int divider
div

- The pipelined machine supports multiple outstanding FP operations:
  - Up to four outstanding FP adds
  - Up to seven outstanding FP/integer multiplies
  - One divide since divide is not pipelined
- Complications :
  - Requires introduction of additional pipeline registers (e.g. a1, a2, a3, a4)
  - The id/ex register must be expanded to connect id to ex, div, m1, and a1

Pipeline timing of a set of independent FP operations

MULT	IF	ID	M1	M2	M3	M4	M5	M6	M7	ME	WB
ADD		IF	ID	A1	A2	A3	A4	ME	WB		
LD			IF	ID	EX	ME	WB				
SD				IF	ID	EX	ME	WB			

Stall occurs when the ADD depends on MULT

MULT	IF	ID	M1	M2	M3	M4	M5	M6	M7	ME	WB
ADD		IF	ID	A1 stall	A2 stall	A3 stall	A4 stall	ME stall	WB		
ADD		IF	ID	stall	stall	stall	stall	stall	A1	A2	A3

Advanced Pipelining, ILP, and Loop-level Parallelism

- We have seen how pipelining can overlap the execution of instructions when they are independent of one another
- This is called Instruction Level Parallelism (ILP) since the instructions can be evaluated in parallel
- Now we will look at how the pipelining ideas can be extended by increasing amount of parallelism exploited among instructions
- Amount of parallelism available within a basic block (i.e. a straight line code sequence with no branches in except to the entry and no branches out except at the exit) is small

One approach is to increase amount of parallelism among iterations of a loop called **loop-level parallelism**

```
for     ( i = 1, i <= 1000, i = i +1 )
        x[i] = x[i] + y[i] ;
```

Every iteration of the loop can overlap with any other iteration (i.e. **loop level parallelism** exists) but within each loop iteration there is not much overlap

We are interested in increasing **instruction level parallelism** in loops like above

- To keep a pipeline full, parallelism among instructions must be exploited by finding sequences of unrelated instructions that can be overlapped in pipeline
- To avoid pipeline stalls, a dependent instruction must be separated from the source instruction by a distance in clock cycles equal to the pipeline latency of that source instruction
- How well this can be done by compiler depends on the ILP available in the program and on the latencies of the functional units
- One key is to **hide** stalls

We make up the following latencies:

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

Assume integer load latency of 1, an integer ALU operations latency of 0, a latency of 1 for branch

- We will look at how compiler (or may be you !) can increase the amount of ILP by **unrolling** loops

```
for ( i =1; i <= 1000 ; i ++ )
        x[i] = x[i] + s ;
```

- The loop is parallel since each iteration is independent
- But we are interested in **ILP** (not loop-level parallelism) now

- Straightforward assembly code for the loop with latencies shown in table previously :

```
loop :   LD      F0, 0(R1)     ;F0 Array element
         ADDD    F4, F0, F2    ;add scalar in F2
         SD      0(R1), F4     ;store result
         SUBI    R1, R1, 8     ;decrement pointer
                                8 bytes
         BNEZ    R1, loop      ;branch R1!=zero
```

- R1 is initially the address of the element in the array with the highest address, and F2 contains the scalar value s; we assume that the element with lowest address is at zero

Look at the execution of the loop without scheduling

			clock cycle issued
loop:	LD	F0, 0(R1)	1
	stall		2
	ADDD	F4, F0, F2	3
	stall		4
	stall		5
	SD	0(R1), F4	6
	SUBI	R1,R1, #8	7
	stall		8
	BNEZ	R1,loop	9
	stall		10

This requires 10 clock cycles per iteration; 1 stall for LD,2 for the ADDD, 1 for SUBI, and one for branch

Look at the execution of the loop with scheduling by compiler

			clock cycle issued
loop:	LD	F0, 0(R1)	1
	SUBI	R1,R1, #8	2
	ADDD	F4, F0, F2	3
	stall		4
	BNEZ	R1,loop	5
	SD	8(R1), F4	6

Observations from scheduled loop

- Execution time has been reduced from 10 clock cycle (without scheduling) to 6 clock cycle (with scheduling)
- One loop iteration completed and result stored back every 6 clock cycle but actual operations (load, add, and store) takes 3 of those clock cycles
- Remaining three clock cycles required for loop overhead, SUBI, BNEZ, and a stall
- Question : How can we decrease the ratio of loop overhead ?

- Answer : Add more non-loop overhead i.e. load, add and store operations within a loop
- A scheme for increasing the number of instructions relative to the number of branch and overhead instructions is **loop unrolling**
- Loop unrolled four times :

```
for ( i = 1; i <= 1000 ; i = i  + 4 ) {
      x[i]       =  x[i] + s ;
      x[i + 1] =  x[i + 1] + s ;
      x[i + 2] =  x[i + 2] + s ;
      x[i + 3] =  x[i + 3] + s ;
  }
```

Unrolled loops cycles without scheduling

Loop :		clock cycle issued	
LD	F0 , 0(R1)	1	
ADDD	F4,F0,F2	2	
SD	0(R1), F4	3	(drop SUBI & BNEZ)
LD	F6,-8(R1)	4	
ADDD	F8,F6, F2	5	
SD	-8(R1), F8	6	(drop SUBI & BNEZ)
LD	F10, -16(R1)	7	
ADDD	F12, F10, F2	8	
SD	-16(R1), F12	9	(drop SUBI & BNEZ)
LD	F14, -24(R1)	10	
ADDD	F16, F14, F2	11	
SD	-24(R1), F16	12	
SUBI	R1, R1, #32	13	
BNEZ	R1,loop	14	

- This will run in **28 cycles –** each LD has 1 stall, each ADDD 2, SUBI 1, the branch 1, plus 14 instruction issue cycles
- **7 clock cycles** for **each** of the four elements

Unrolled loops cycles with scheduling

```
loop:   LD      F0, 0 (R1)
        LD      F6, - 8 (R1)
        LD      F10, -16 (R1)
        LD      F14, -24 (R1)
        ADDD    F4, F0, F2
        ADDD    F8, F6, F2
        ADDD    F12, F10, F2
        ADDD    F16, F14, F2
        SD      0(R1), F4
        SD      -8 (R1), F8
        SUBI    R1, R1, #32
        SD      -16 (R1),F12
        BNEZ    R1, loop
        SD      8(R1), F16      ; (-32 + 8 = -24)
```

- Execution time of the unrolled loop has dropped to a **total of 14 clock cycles**, or **3.5 clock cycles per element**, compared with 7 cycles per element before scheduling and 6 cycles when scheduled but not unrolled

Benefits of unrolling

- Gain from scheduling on the unrolled loop is even larger than on the original loop
- Unrolling loops exposes more computation that can be scheduled to minimize stalls
- Loop unrolling is a simple but useful method for increasing the size of straight line code fragments that can be scheduled effectively

Summary of loop unrolling

- Determine that loop unroll would be useful by finding that the loop iterations were independent, except for the loop maintenance code
- Use different registers to avoid constraints that would be needed if the same registers were used for the same computations
- Eliminate the extra tests and branches and adjust loop maintenance code
- Determine that loads and stores in the unrolled can be interchanged since the loads and stores from different iterations are independent
- Schedule the code so that any dependency needed to yield the same result at the original code is preserved

Loop-level Parallelism

- Loop-level parallelism is normally analyzed at the source level or close to it; while most analysis of ILP is done once instructions have been generated by the compiler
- Loop-level analysis involves determining what dependencies exist among the operands in the loop across the iterations of the loop
- We will concentrate on data dependencies which arise when an operand is written at some point and read at a later point
- Need to determine whether data accesses in later iterations are data dependent on data values produced in earlier iterations
- Dependence in the following loop body between two uses of x[i], but this dependence is within a single iteration. No dependence between instructions in different iterations

```
for  ( i = 1 ; i <= 1000; i ++)
        x[i] = x[i] + s ;
```

```
for  (i = 1; i <= 100; i = i + 1) {
A [i + 1] = A [i] + C [i] ;          /* S1*/
B [i + 1] = B [i] + A[i + 1]      /* S2 */ }
```

- There are two different kind of dependencies :
  - S1 uses a value computed by S1 in an earlier iteration since iteration i computes A[i +1] which is read in iteration i +1; same is true of S2 for B[i] and B[i+1]
  - Another dependency is that S2 uses A [i+1] which is computed by S1 in the same iteration
- The dependence of S1 on an earlier iteration of S1 is called **loop-carried** dependence implying dependence exists between different iterations of the loop
- The second dependence (i.e. S2 depending on S1) is not loop-carried and if existed alone would not prevent execution of multiple iterations in parallel

- ```
  for ( i = 1; i < = 100 ; i = i +1 ) {
      A[i] = A[i] + B[i] ;      /* S1 */
      B[i+1] = C[i] + D[i];   /* S2 */}
  ```
- Statement S1 uses the value assigned in the previous iteration by statement S2, so there is **loop carried dependency** between S2 and S1
- S1 depends on S2 but S2 does not depend on S1, so we can interchange them
- Neither statement depends on itself
- On the first iteration of the loop, S1 depends on B[1]

```
A[1] = A[1] + B[1]
for ( i = 1; i < = 99 ; i = i +1 ) {
    B[i+1] = C[i] + D[i] ;          /* S1 */
    A[i+1] = A[i+1] + B[i+1];      /* S2 */}
B[101] = C[100] + D[100];
```

- No more loop-carried dependence
```

ILP versus Reducing Cache Misses

- Sometimes compiler must choose between improving ILP and improving cache misses

```
for ( i = 0; i < 512; i = i + 1 )
        for ( j = 1, j < 512 ; j = j + 1 )
                x[i][j] = 2* x[i][j-1] ;
```

- Data is accessed in order they are stored (row wise in C), hence minimizes cache misses

To increase ILP we try to unroll the loop :

```
for ( i = 0; i < 512 ; i = i +1 )
    for ( j = 1; j < 512; j= j + 4) {
          x[i][j]     = 2*x[i][j-1] ;
          x[i][j+1] = 2*x[i][j] ;
          x[i][j+2] = 2*x[i][j+1] ;
          x[i][j+3] = 2*x[i][j+2] ;
    }
```

Data dependency prevents loop unrolling for ILP