Loop Unrolling

Pipelining :Loop Unrolling

Latencies of FP operations used in the example		
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**

Pipelining :Loop Unrolling Example

- for (i=1; i<=1000; i++) x[i] = x[i] + s;
- Where X is an array
  s is a
constant Assume
- R1 contains address of an array X
- R2 has termination address
- F2 has constant s.

Pipelining Loop Unrolling: Assembly code

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 (per double)**
	BENZ	**R1, Loop**	**;branch R1 != zero**

Without unrolling and without scheduling

			Cycles
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**
	BENZ	**R1, Loop**	**8**
	stall		**9**

9 clock cycles per iteration

Without unrolling and with scheduling

			Cycles	
Loop:	LD	F0, 0(R1)	1	
	SUBI	R1, R1, #8	2	
	ADDD	F4, F0, F2	3	
	Stall		4	
	BENZ	R1,Loop	5	;delayed branch
	SD	8(R1), F4	6	;altered and interchanged with SUBI
6 clock cycles per element				

Loop Unrolling: 4 iterations

Loop:	LD	F0, 0(R1)		
	ADDD	F4, F0, F2		
	SD	0(R1), F4	#1	
	LD	F6, -8(R1)		
	ADDD	F8, F6, F2		
	SD	-8(R1), F8	#2	
	LD	F10,-16(R1)		
	ADDD	F12,F10,F2		
	SD	-16(R1), F12	#3	
	LD	F14,-24(R1)		
	ADDD	F16,F14,F2		
	SD	-24(R1),F16	#4	
	SUBI	R1, R1, #32		
	BENZ	R1, Loop		

Loop Unrolling :Without any

Loop:	LD	F0, 0(R1)	1	
	stall		2	
	ADDD	F4, F0, F2	3	
	stall		4	
	stall		5	
	SD	0(R1), F4	6	;drop SUBI &BNEZ #1
	LD	F6, -8(R1)	7	
	stall		8	
	ADDD	F8, F6, F2	9	
	stall		10	
	stall		11	
	SD	-8(R1), F8	12	;drop SUBI &BNEZ #2
	LD	F10,-16(R1)	13	
	stall		14	
	ADDD	F12,F10,F2	15	
	stall		16	
	stall		17	
	SD	-16(R1), F12	18	;drop SUBI &BNEZ #3
	LD	F14,-24(R1)	19	
	stall		20	
	ADDD	F16,F14,F2	21	
	stall		22	
	stall		23	
	SD	-24(R1),F16	24	#4
	SUBI	R1, R1, #32	25	
	BENZ	R1, Loop	26	
	Stall		27	

Loop Unrolling :Without any scheduling

- It takes 27 cycles for 4 iterations

  27/4 = 6.8 clock cycles per iteration

- CPI =6.8/3 =2.2

Loop Unrolling :With scheduling

	Instruction		cycles	
Loop:	**LD**	**F0, 0(R1)**	**1**	
	LD	**F6, -8(R1)**	**2**	
	LD	**F10,-16(R1)**	**3**	
	LD	**F14,-24(R1)**	**4**	
	ADDD	**F4, F0, F2**	**5**	
	ADDD	**F8, F6, F2**	**6**	
	ADDD	**F8, F6, F2**	**7**	
	ADDD	**F16, F14, F2**	**8**	
	SD	**0(R1), F4**	**9**	
	SD	**-8(R1), F8**	**10**	
	SD	**-16(R1), F12**	**11**	
	SUBI	**R1, R1, #32**	**12**	
	BENZ	**R1, Loop**	**13**	
	SD	**8(R1), F16**	**14**	**;8-32=-24**

Loop Unrolling :With scheduling

- 14 clock cycles per 4 iterations 14/4 = 3.5 clock cycles per iteration

- CPI=3.5/3 =1.16

Loop unrolling

Loop unrolling is a technique that seeks to ensure you do a reasonable number of data operations for the overhead of running through a loop. Take the following code:

```
{
for (i=0;i<100;i++)
q[i]=i;
}
```

Loop unrolling

- In terms of assembly code, this will generate:
- A load of a register with 0 for parameter i.
- A test of the register with 100.
- A branch to either exit or execute the loop.
- An increment of the register holding the loop counter.
- An address calculation of array q indexed by i.
- A store of i to the calculated address.
- Only the last of these instructions actually does some real work.
- The rest of the instructions are overhead

Loop unrolling

We can rewrite this C code as

```
{
for (i=0;i<25;i+=4)
        q[i]=i;
        q[i+1]=i+1;
        q[i+2]=i+2;
        q[i+3]=i+3;
}
```

Loop invariant analysis

Loop invariant analysis looks for expressions that are constant within the loop body and moves them outside the loop body.

Example:

```
for (int j=0;j<100;j++)
{
for (int i=0; i<100; i++)
{
const int b = j * 200;
q[i]=b;
}
}
```

Loop invariant analysis

- The parameter j is constant within the loop body for parameter i.
- Thus, the compiler can easily detect this and will move the calculation of b outside the inner loop.

Loop invariant analysis

```
for (int j=0;j<100;j++)
{
const int b = j * 200;
for (int i=0; i<100; i++)
{
q[i]= b;
}
}
```

This optimized code removes thousands of unnecessary calculations of b, where j, and thus b, are constant in the inner loop.