Lesson 08:
Compilation Techniques and Support to Instruction Level Parallelism
Objective

- To learn compilation for supporting performance improvement by extracting parallelism
- Loop unrolling and software pipelining
Instruction-level parallel processor
Techniques used by Compilers for supporting performance improvement

- Constant propagation
- Dead-code elimination
- Register allocation
- Loop unrolling, an optimization that significantly increases instruction-level parallelism
Extracting Parallelism by loop unrolling
Loop unrolling

- Branches result in branch penalty due to control hazard
- Loop unrolling addresses the limitation of limited number of instructions between two branches
Loop unrolling

- Transforming a loop with $N$ iterations into a loop with $N/M$ iterations
- Each of the iterations in the new loop does the work of $M$ iterations of the old loop
C program Loop Unrolling Example

```
Original Loop

for (i = 0; i < 100; i++) {
    a[i] = b[i] + c[i];
}
```
C program Loop Unrolling Example for running in parallel on two pipelines

```c
for (i = 0; i < 100; i += 2) {
    a[i] = b[i] + c[i];
    a[i+1] = b[i+1] + c[i+1];
}
```
Loop unrolling

- Increases the number of instructions between branches
- Giving the compiler and the hardware more opportunity to find instruction-level parallelism.
Arranging instructions

- Independent instructions are close together in the program
- Placing the pointer increments between the loads and the computation of $a[i]$
Arranging instructions

• Increase the number of operations between the loads and the use of their results, making it more likely that the loads will complete before their results are needed
Assembly program Loop Unrolling Example

Original Loop

MOV r31, #0; /* initialize loop count*/
loop: LD r1, (r2);  /*r2 is pointer to b[i]*/
LD r3, (r4);  /*r4 is pointer to c[i]*/
ADD r2, #4, r2;  /*increment to b[i + 1]*/
ADD r4, #4, r2;  /*increment to c[i + 1]*/
ADD r6, r1, r3;
ST (r7), r6;  /*r7 is pointer to a[i]*/
ADD r7, #4, r7;  /*r7 is pointer to a[i + 1]*/
ADD r31,, #1, r31;  /*increment count*/
BNE loop, #r31, #r100;  /*jump next iteration till count=100 times only*/
Assembly program Loop Unrolling

Unrolled Loop

```
MOV r_{31}, #0; /* initialize loop count*/
ADD r_8, #4, r_2; /* inc. r_8, point b[i+1]*/
ADD r_{10}, #4 r_4; /* increment r_{10}, c[i+1]*/
ADD r_{13}, #4 r_6; /* increment r_{13}, a[i+1]*/
loop: LD r_1, (r_2); /* r_1 loads b[i]*/
LD r_9, (r_8); /* r_9 loads b[i+1]*/
LD r_3, (r_4); /* r_3 loads c[i]*/
LD r_{11}, (r_{10}); /* r_{11} loads c[i+1]*/
```
Assembly program Loop Unrolling

```
ADD r₂, #8, r₂; /*increment r₈, b[i+2]*/
ADD r₄, #8, r₄; /*increment to c[i+2]*/
ADD r₈, #8, r₈; /*increment to b[i+3]*/
ADD r₁₀, #8, r₁₀; /*increment to c[i+3]*/
ADD r₆, r₁, r₃; /*Add into a the b, c iᵗʰ int*/
ADD r₁₂, r₉, r₁₁; /*Add i + 1ᵗʰ integer*/
ST (r₇), r₆;     /*stores a[i]*/
ST (r₁₃), r₁₂; /*stores a[i+1]*/
ADD r₇, #8, r₇; /*r₇ pointer to a[i+2]*/
ADD r₁₃, #8, r₁₃; /*r₇ pointer to a[i+3]*/
ADD r₃₁,, #2, r₃₁; /*increment count by 2*/
```
Unrolled loop

- Begins with three adds to generate pointers to $a[i + 1]$, $b[i + 1]$, and $c[i + 1]$.
- Keep these pointers in separate registers from the pointers to $a[i]$, $b[i]$, and $c[i]$
Unrolled loop

- It allows the loads and stores to the $i^{th}$ and $(i + 7)^{th}$ elements of each array to be done in parallel, rather than having to increment each pointer between memory references.
Uneven Loop Unrolling
C program Loop Unrolling Example

Original Loop

```c
for (i = 0; i < 100; i++) {
    a[i] = b[i] + c[i];
}
```
C program first Loop after Unrolling

Unrolled Loop
for (i=0; i<100; i += 8) {
    a[i] = b[i] + c[i];
    a[i+1] = b[i+1] + c[i+1];
    a[i+2] = b[i+2] + c[i+2];
    a[i+3] = b[i+3] + c[i+3];
    a[i+4] = b[i+4] + c[i+4];
    a[i+5] = b[i+5] + c[i+5];
    a[i+6] = b[i+6] + c[i+6];
    a[i+7] = b[i+7] + c[i+7];
}
for (i = ((100/8) × 8); i < 100; 
    i++) {
    a[i] = b[i] + c[i];
}
First loop unrolled eight times

- The first loop steps through the iterations eight at a time, until there are fewer than eight iterations remaining (detected when \( i + 8 \geq 100 \)).
Second loop steps through the remaining iterations one at a time

- The second loop starts at the next iteration when fewer than 8 iterations remained at the first loop.
- Because $i$ is an integer variable, the computation $i = ((100/8) \times 8)$ does not set $i$ to 100.
- It executes for 96, 97, 98 and 99.
Software Pipelining
Software Pipelining by Interleaving portions of different loop iterations

- Improves performance by distributing each of iterations of original loop over multiple iterations of the pipelined loop
- Each iteration of the new loop performs some of the work of multiple iterations of the original loop
Interleaving portions of different loop iterations

- Increases instruction-level parallelism in much the same way that loop unrolling does.
- Increases the number of instructions between the computation of a value and its use, making it more likely that the value will be ready before it is needed.
Example

- Transform a loop that fetched $b[i]$ and $c[i]$ from memory, added them together to generate $a[i]$ and wrote $a[i]$ back to memory by using software pipelining
Solution

- Each interaction first wrote $a[i - 1]$ back to memory, then computed $a[i]$ based on the values of $b[i]$ and $c[i]$ that were fetched in the last iteration, and finally fetched $b[i + 1]$ and $c[i + 1]$ from memory to prepare for the next iteration.
Solution

• Thus, the work of computing a given element of the $a[ ]$ array is distributed across three iterations of the new loop
Combination of loop unrolling and Software Pipelining
Combination of loop unrolling and Software Pipelining

- Many compilers combine software pipelining and loop unrolling to increase instruction-level parallelism further than is possible by applying either optimization individually.
Summary
We Learnt

- Extracting of parallelism by loop unrolling
- Loop unrolling increased number of instructions per loop, thus less number of control hazards
- Loop unrolling enabled division of work to multiple pipelines in place of one pipeline executing the full loop
We Learnt

- Software pipelining by distributing each iteration over multiple iterations of the pipelined loop
End of Lesson 06 on Compilation Techniques and Support to Instruction Level Parallelism