Computer Organization and Architecture

Instructor: Dr. Rushdi Abu Zneit

Slide Sources: Based on CA: aQA by Hennessy/Patterson.

Advanced Topic: Compiler Support for ILP CA:aQA Sec. 4.1

Scheduling Code for the MIPS Pipeline

- Example:

  -
    ```
    for (i=1000; i>0; i=i-1)
          x[i] = x[i] + s;
    ```
- Notes:
  - the loop is *parallel* – the body of each iteration is *independent* of that of other iterations
  - *conceptually* : if we had 1000 CPUs, we could distribute one iteration to each CPU and compute in parallel (=*simultaneously*)
- *Only the compiler can exploit such instruction-level parallelism (ILP), not the hardware!* Why?
  - because only the compiler has a *global view* of the code
  - the hardware sees each line of code only after it is fetched from memory, *not all together* – in particular, *not the whole loop*
  - the compiler must schedule the code intelligently to expose and exploit ILP…

Scheduling Code for the MIPS Pipeline

- Assume *FP operation* latencies as below
  - *latency* indicates number of intervening cycles required between producing and consuming instruction to avoid stall
- Assume *integer ALU operation* latency of 0 and integer load latency of 1

FP Latency table

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

Unscheduled Code

Original C loop statement: `for (i=1000; i>0; i=i-1) x[i] = x[i] + s;`

Unscheduled code for the MIPS pipeline:

```
Loop:    L.D       F0,0(R1)      ;F0 = array element
         ADD.D     F4,F0,F2      ;add scalar in F2
         S.D       F4,0(R1)      ;store result
         DADDUI    R1,R1,#-8     ;decrement pointer
                                 ;8 bytes per DW
         BNE       R1,R2,Loop    ;branch R1!=R2
```

Execution cycles for the unscheduled code:

```
                                     Clock cycle issued
Loop:    L.D       F0,0(R1)                 1
         stall                              2
         ADD.D     F4,F0,F2                 3
         stall                              4
         stall                              5
         S.D       F4,0(R1)                 6
         DADDUI    R1,R1,#-8                7
         stall                              8
         BNE       R1,R2,Loop               9
         stall                             10
```

Why one stall? Think of when the optimized MIPS pipeline resolves branch outcomes…

Delayed branch stall

10 clock cycles per iteration

Scheduled Code

Scheduled code for the MIPS pipeline:

```
Loop:    L.D      F0,0(R1)
         DADDUI   R1,R1,#-8
         ADD.D    F4,F0,F2
         BNE      R1,R2,Loop    ;delayed branch
         S.D      F4,8(R1)      ;altered and interchanged with DADDUI
```

Execution cycles for the scheduled code:

```
                                    Clock cycle issued
Loop:    L.D        F0,0(R1)                1
         DADDUI     R1,R1,#-8               2
         ADD.D      F4,F0,F2                3
         stall                              4
         BNE        R1,R2,Loop              5
         S.D        F4,8(R1)                6
```

6 clock cycles per iteration is optimal because of the dependencies. Only 3 of the operations (`L.D`, `ADD.D` & `S.D`) actually operate on the array, the other three are loop overhead…

- Compiler has to be "smart" to perform this scheduling
  - e.g., interchanging the `DADDUI` and `S.D` instructions requires understanding the dependence between them and accordingly changing the `S.D` store address from `0(R1)` to `8(R1)`!

Unrolling Loops

- The 3 clock cycle per iteration overhead delay in the scheduled code of the previous example may be reduced…
  - …by *amortizing* the loop overhead over multiple loop iterations
- For this we need to *unroll the loop* and block multiple iterations into one
- Loop unrolling also allows *improved scheduling* by exposing increased ILP – between instruction from different iterations
- Example…

Unrolling Loops – High-level

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

C equivalent of unrolling to block four iterations into one:

```
for (i=250; i>0; i=i-1)
{
   x[4*i]   = x[4*i]   + s;
   x[4*i-1] = x[4*i-1] + s;
   x[4*i-2] = x[4*i-2] + s;
   x[4*i-3] = x[4*i-3] + s;
}
```

Unrolled Loop – not Scheduled

Unrolled but unscheduled code for the MIPS pipeline:

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

- Notes:
  - four copies of the loop body have been unrolled
  - different registers are used in each copy – to facilitate future scheduling
  - three branches and three decrements of `R1` have been eliminated

Executing the Unrolled Unscheduled Loop

```
                                          Clock cycle issued
Loop:    L.D       F0,0(R1)                        1
         stall                                     2
         ADD.D     F4,F0,F2                        3
         stall                                     4
         stall                                     5
         S.D       F4,0(R1)                        6
         L.D       F6,-8(R1)                       7
         stall                                     8
         ADD.D     F8,F6,F2                        9
         stall                                    10
         stall                                    11
         S.D       F8,-8(R1)                      12
         L.D       F10,-16(R1)                    13
         stall                                    14
         ADD.D     F12,F10,F2                     15
         stall                                    16
         stall                                    17
         S.D       F12,-16(R1)                    18
         L.D       F14,-24(R1)                    19
         stall                                    20
         ADD.D     F16,F14,F2                     21
         stall                                    22
         stall                                    23
         S.D       F16,-24(R1)                    24
         DADDUI    R1,R1,#-32                     25
         stall                                    26
         BNE       R1,R2,Loop                     27
         stall                                    28
```

One iteration of the unrolled loop runs in 28 clock cycles. Therefore,
7 clock cycles per iteration of original loop – slower than scheduled original loop!

Scheduling the Unrolled Loop

Unrolled and scheduled code for the MIPS pipeline:

```
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,-24(R1)   ;8-32 = -24
```

Executing the Unrolled and Scheduled Loop

```
                                          Clock cycle issued
Loop:    L.D      F0,0(R1)                       1
         L.D      F6,-8(R1)                      2
         L.D      F10,-16(R1)                    3
         L.D      F14,-24(R1)                    4
         ADD.D    F4,F0,F2                       5
         ADD.D    F8,F6,F2                       6
         ADD.D    F12,F10,F2                     7
         ADD.D    F16,F14,F2                     8
         S.D      F4,0(R1)                       9
         S.D      F8,-8(R1)                     10
         DADDUI   R1,R1,#-32                    11
         S.D      F12,-16(R1)                   12
         BNE      R1,R2,Loop                    13
         S.D      F16,-24(R1)                   14
```

No stalls! One iteration of the unrolled loop runs in 14 clock cycles. Therefore,
3.5 clock cycles per iteration of original loop vs. 6 cycles for scheduled but not unrolled loop

Notes

- Scheduling code (if possible) to avoid stalls is always a win and optimizing compilers typically generate scheduled assembly
- Unrolling loops can be advantageous but there are potential problems
    - *growth in code size*
    - *register pressure*: aggressive unrolling and scheduling requires allocation of multiple registers

Enhancing Loop-Level Parallelism

- Consider the previous running example:

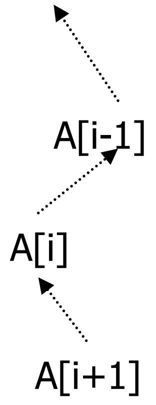
  -
    ```
    for (i=1000; i>0; i=i-1)
       x[i] = x[i] + s;
    ```
  - there is no *loop-carried dependence* – where data used in a later iteration depends on data produced in an earlier one
  - in other words, all iterations could (conceptually) be executed in parallel
- Contrast with the following loop:

  -
    ```
    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 */
    }
    ```
  - *what are the dependences?*

A Loop with Dependences

- For the loop:

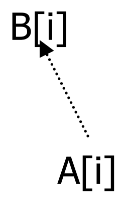
  -
    ```
    for (i=1; i<=100; i=i+1) {
      A[i+1] = A[i] + C[i];       /* S1 */
      B[i+1] = B[i] + A[i+1];     /* S2 */
    }
    ```
  - *what are the dependences?*
- There are two different dependences:
  - loop-carried:
    - S1 computes `A[i+1]` using value of `A[i]` computed in previous iteration
    - S2 computes `B[i+1]` using value of `B[i]` computed in previous iteration
  - not loop-carried:
    - S2 uses the value `A[i+1]` computed by S1 in the *same* iteration
- The loop-carried dependences in this case *force* successive iterations of the loop to execute in series. *Why?*
  - S1 of iteration *i* depends on S1 of iteration *i-1* which in turn depends on …, etc.



Another Loop with Dependences

- Generally, loop-carried dependences *hinder* ILP
  - if there are no loop-carried dependences all iterations could be executed in parallel
  - even if there are loop-carried dependences it may be possible to parallelize the loop – an analysis of the dependences is required…
- For the loop:

  -
    ```
    for (i=1; i<=100; i=i+1) {
       A[i] = A[i] + B[i];       /* S1 */
       B[i+1] = C[i] + D[i];     /* S2 */
    }
    ```
  - *what are the dependences?*
- There is one loop-carried dependence:
  - S1 uses the value of `B[i]` computed in a previous iteration by S2
  - but this *does not force* iterations to execute in series. *Why…?*
  - …because S1 of iteration *i* depends on S2 of iteration *i-1*…, and the *chain of dependences stops here*!



Parallelizing Loops with Short Chains of Dependences

- Parallelize the loop:

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

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

  - the dependence between the two statements in the loop is no longer loop-carried and iterations of the loop may be executed in parallel

Another Example

- Analyze the loop:

  -
    ```
    for (i=1; i<=100; i=i+1) {
       A[i] = A[i] + B[i];         /* S1 */
       B[i+1] = C[i] + D[i];       /* S2 */
       C[i+1] = E[i] + D[i];       /* S3 */
    }
    ```