ELE 655
Microprocessor System Design

Section 2 – Instruction Level Parallelism

Class 2 – ILP

ILP

Instruction level Parallelism

- ILP
  - Exploit parallelism among instructions
  - Basic Block
    - Section of linear code with no branches (calls, …)
    - Typical instruction run length (basic block) is 3-6 instructions
    - Usually have high dependence within the block

    → very difficult to exploit any parallelism here
  - Focus on ILP between multiple basic blocks

ILP

Instruction level Parallelism

- Loops
  - Each iteration contains a basic block
  - Look for opportunities among iterations of the loop
  - Loop level parallelism

```
For (i=0; i<=999; i=i+1)
    x[i] = x[i] + y[i];
```

- No basic block opportunity
- How can we parallelize the loop?

ILP
Instruction level Parallelism

- Data Dependences
  - One instruction is dependent on the result from a prior instruction
  - If a pipeline is sufficiently deep to cause errors a hazard must be detected
  - The effect of the original sequence must be preserved
    - Can keep the dependency and avoid the hazard (forwarding, …)
    - Eliminate the dependency by modifying the code (compiler. …)

ILP

Instruction level Parallelism

- Data Dependences

  - Register based data dependencies

```
Loop:   L.D      F0,0(R1)     ; F0 = array element
        ADD.D    F4,F0,F2     ; add scalar to F2
        S.D      F4,0(R1)     ; store result
```

  - Easy to detect ADD depends on L.D via register F0

ILP

Instruction level Parallelism

- Data Dependences
  - Memory based data dependencies
  - What if R1=0x32 and R2 = 0x40
    0(R2) → 0x40   AND   8(R1) → 0x40

    We have a hazard !
  - Also note 4(R4) may not point to the same memory location each iteration since R4 could change

ILP

Instruction level Parallelism

- Name Dependences

  - Look like data dependencies BUT no data is transferred between instructions

  - Antidependence (i,j)
    - j writes a register i reads
    - Order must be preserved

```
S.D       F4,0(R1)
DADDUI    R1,R1,#-8
```

  - Output Dependence (i,j)
    - i and j write the same register or memory location
    - Order must be preserved

ILP

Instruction level Parallelism

- Name Dependences
  - Since these are not true data dependences
    - Reordering is possible as long as program order is maintained
    - Register renaming
      - Compile time
      - Execute time

ILP

Instruction level Parallelism

- Data Hazards
  - Name or Data dependence AND overlap in the pipeline
  - SW and HW techniques seek to maintain program order ONLY when failure to do so will change the result

ILP

Instruction level Parallelism

- Data Hazards
  - Instruction i followed by instruction j

  - RAW – read after write
    - j ties to read a value before i writes it
    - True data dependence

  - WAW – write after write
    - j tries to write a value before i writes it – final value is i instead of j
    - Output dependence
    - Requires multiple stages to write

  - WAR – write after read
    - j tries to write a value before i reads it –i gets j instead of original value
    - Antidependence
    - Uncommon

ILP
Instruction level Parallelism

- Control Dependences
  - Branch dependences –

```
If p1 {
        s1;         dependent on p1
};
If p2 {
        s2;         dependent on p2, but not p1
};
```

  - An instruction that is dependent on a branch cannot be moved **before** the branch if the movement removes the dependence on the branch
  - An instruction that is not dependent cannot be moved **after** the branch if the movement creates a dependence

ILP

Instruction level Parallelism

- Control Dependences
  - Any reordering of instructions must
    - Preserve exception behavior

```
        DADDU    R2,R3,R4
        BEQZ     R2,L1
        LW       R1,0(R2)
L1:     …
```

- No data dependence between BEQZ and LW (at least in this snippit)
- Control dependence? - possible memory protection exception

ILP
Instruction level Parallelism

- Control Dependences
  - Any reordering of instructions must
    - Preserve data flow (and correctness) – not just dependence
    - Branches allow data to come from multiple locations

```
        DADDU       R1,R2,R3
        BEQZ        R4,L
        DSUBU       R1,R5,R6
L:      …
        OR          R7,R1,R8
```

    - Can maintain data dependence on R1 – data will be available in time either way
    - Order can be preserved
    - Data flow changes depending on branch status

ILP

Instruction level Parallelism

- Control Dependences
  - Any reordering of instructions must
    - Preserve exception behavior
    - Preserve data flow (and correctness) – not just dependence

```
            DADDU       R1,R2,R3
            BEQZ        R12, skip
            DSUBU       R4,R5,R6
            DADDU       R5,R4,R9  ; R4 unused after this instruction – R4 is DEAD
skip:       OR          R7,R8,R9
```

- Compiler could reorder the DSUBU to before the branch if it was useful to us
  - Original - Betting the branch is usually not taken – why???
  - Modified – Betting the branch is usually taken
- Note – DEAD implies not used in its current state, not unused at all

ILP
Instruction level Parallelism

- Pipeline Scheduling

  - To avoid a stall
    - Execution of dependent instructions must be spaced
    - Spacing depends on latencies of instructions

    - Latencies – for our examples

Inst producing result	Inst using result	Latency (clock cycles)
FP ALU op	FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store Double	0
Branches		1

ILP
Instruction level Parallelism

- Pipeline Scheduling

```
for (i=999; i>=0; i=i+1)
        x[i] = x[i] + s;
```

- Note: body of each iteration is independent → opportunities to parallelize

- Compiles to

```
loop:   L.D      F0,0(R1)      ; F0 array element
        ADD.D    F4,F0,F2      ; F2 holds scalar
        S.D      F4,0(R1)      ; store result
        DADDUI   R1,R1,#-8     ; decrement pointer
        BNE      R1,R2,loop    ; loop until R1=R2
```

ILP
Instruction level Parallelism

- Pipeline Scheduling
  - No Scheduling

```
                                        clock cycles
loop:     L.D       F0,0(R1)                 1
              stall                          1
          ADD.D     F4,F0,F2                 1
              stall                          1
              stall                          1
          S.D       F4,0(R1)                 1
          DADDUI    R1,R1,#-8                1
              stall                          1
          BNE       R1,R2,loop               1
                                        __________
                                             9
```

ILP

Instruction level Parallelism

- Pipeline Scheduling
  - With Scheduling
    - Note DADDUI is independent of anything else in loop
    - Moving it also clears up the BNE data dependence

```
                                          clock cycles
loop:     L.D        F0,0(R1)                  1
          DADDUI     R1,R1,#-8                 1
          ADD.D      F4,F0,F2                  1
               stall                           1
               stall                           1
          S.D        F4,0(R1)                  1
          BNE        R1,R2,loop                1
                                          ___________
                                               7
```

- 22% improvement

ILP

Instruction level Parallelism

- Pipeline Scheduling
  - With Scheduling
    - Here 3 clock cycles are actually working on the data
      - Load, add, store
    - 4 clock cycles are attached to the loop overhead
    - No room for improvement on the data
    - With 1000 iterations – can we reduce the loop overhead?

ILP
Instruction level Parallelism

- Pipeline Scheduling
  - Loop unrolling
    - Attempt to reduce the loop overhead by combining multiple iterations into a single iteration

```
loop:       L.D       F0,0(R1)       ; F0 array element
            ADD.D     F4,F0,F2       ; F2 holds scalar
            S.D       F4,0(R1)       ; store result
            DADDUI    R1,R1,#-8      ; decrement pointer
            L.D       F0,0(R1)       ; F0 array element
            ADD.D     F4,F0,F2       ; F2 holds scalar
            S.D       F4,0(R1)       ; store result
            DADDUI    R1,R1,#-8      ; decrement pointer
            L.D       F0,0(R1)       ; F0 array element
            ADD.D     F4,F0,F2       ; F2 holds scalar
            S.D       F4,0(R1)       ; store result
            DADDUI    R1,R1,#-8      ; decrement pointer
            L.D       F0,0(R1)       ; F0 array element
            ADD.D     F4,F0,F2       ; F2 holds scalar
            S.D       F4,0(R1)       ; store result
            DADDUI    R1,R1,#-8      ; decrement pointer
            BNE       R1,R2,loop     ; loop until R1=R2
```

Name Hazards throughout

Need to rename registers

ILP
Instruction level Parallelism

- Pipeline Scheduling

  - Loop unrolling
    - Attempt to reduce the loop overhead by combining multiple iterations into a single iteration

```
loop:     L.D       F0,0(R1)        ; F0 array element
          ADD.D     F4,F0,F2        ; F2 holds scalar
          S.D       F4,0(R1)        ; store result
          DADDUI    R1,R1,#-8       ; decrement pointer
          L.D       F6,0(R1)        ; F0 array element
          ADD.D     F8,F6,F2        ; F2 holds scalar
          S.D       F8,0(R1)        ; store result
          DADDUI    R1,R1,#-8       ; decrement pointer
          L.D       F10,0(R1)       ; F0 array element
          ADD.D     F12,F10,F2      ; F2 holds scalar
          S.D       F12,0(R1)       ; store result
          DADDUI    R1,R1,#-8       ; decrement pointer
          L.D       F14,0(R1)       ; F0 array element
          ADD.D     F16,F14,F2      ; F2 holds scalar
          S.D       F16,0(R1)       ; store result
          DADDUI    R1,R1,#-8       ; decrement pointer
          BNE       R1,R2,loop      ; loop until R1=R2
```

ILP
Instruction level Parallelism

- Pipeline Scheduling
  - Loop unrolling
    - Attempt to reduce the loop overhead by combining multiple iterations into a single iteration

```
loop:     L.D       F0,0(R1)        ;
          ADD.D     F4,F0,F2        ;
          S.D       F4,0(R1)        ;
          L.D       F6,-8(R1)       ; modify offset
          ADD.D     F8,F6,F2        ;
          S.D       F8,-8(R1)       ;
          L.D       F10,-16(R1)     ; modify offset
          ADD.D     F12,F10,F2      ;
          S.D       F12,-16(R1)     ;
          L.D       F14,-24(R1)     ; modify offset
          ADD.D     F16,F14,F2      ;
          S.D       F16,-24(R1)     ;
          DADDUI    R1,R1,#-32      ; compensate for 4 iterations
          BNE       R1,R2,loop      ; loop until R1=R2
```

Can merge the pointer increment with LD, ST – symbolic substitution and simplification

Still have Data Hazards → scheduling

ILP

Instruction level Parallelism

- Pipeline Scheduling
  - Loop unrolling
    - Attempt to reduce the loop overhead by combining multiple iterations into a single iteration

```
loop:     L.D       F0,0(R1)        ; Clump LDs
          L.D       F6,-8(R1)       ;
          L.D       F10,-16(R1)     ;
          L.D       F14,-24(R1)     ;
          ADD.D     F4,F0,F2        ; Clump Adds
          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      ; avoid the BNE dependence
          S.D       F12,-16(R1)     ;
          S.D       F16,-24(R1)     ;
          BNE       R1,R2,loop      ;
```

Fully Scheduled

ILP

Instruction level Parallelism

- Pipeline Scheduling – loop unrolling

  - Original - 10 clocks/iteration
  - Original – scheduled – 7 clocks/iteration
  - Unrolled - 27 clocks/4 iterations – 6.75 clocks/iteration
  - Unrolled – scheduled – 14 clocks / 4 iterations – 3.5 clocks/iteration

  - 65% performance improvement
  - 14 instructions vs. 5 originally – 2.8x code size

```
loop:     L.D       F0,0(R1)       ; F0 array element
          ADD.D     F4,F0,F2       ; F2 holds scalar
          S.D       F4,0(R1)       ; store result
          DADDUI    R1,R1,#-8      ; decrement pointer
          BNE       R1,R2,loop     ; loop until R1=R2
```

ILP
Instruction level Parallelism

- Pipeline Scheduling – loop unrolling

  - Normally we will not know the upper bound on a loop (n)

  - Create 2 consecutive loops
    - 1st runs (n mod k) iterations of the original loop
    - 2nd runs n/k iterations of the loop unrolled k times

  - How would we do this?

```
loop:     L.D       F0,0(R1)        ; F0 array element
          ADD.D     F4,F0,F2        ; F2 holds scalar
          S.D       F4,0(R1)        ; store result
          DADDUI    R1,R1,#-8       ; decrement pointer
          BNE       R1,R2,loop      ; loop until R1=R2
```

ILP

Instruction level Parallelism

- Pipeline Scheduling – loop unrolling

  - Loop unrolling allows us to create the opportunity for effective scheduling

    1) Determine that the loop contents are independent
    2) Avoid register name hazards
    3) Eliminate extra test and branch instructions
    4) Look for opportunities to modify loads and stores – watching for memory issues
    5) Schedule the code – ensuring any dependencies are preserved

ILP
Instruction level Parallelism

- Pipeline Scheduling – loop unrolling
  - Impacting (limiting) factors
    - Decrease in amount of overhead
      - Bigger unrolls reduce overhead more
    - Code size
      - Cost
      - Impact on instruction cache hit rates
    - Register limitations (register pressure)
      - Increases the number of live registers