CS60003: High Performance Computer Architecture

Instruction-Level Parallelism: Software Techniques

IIT KHARAGPUR

Instructor:
Prof. Rajat Subhra Chakraborty
Professor
Dept. of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Kharagpur, West Bengal, India 721302
E-mail: rschakraborty@cse.iitkgp.ac.in

What is Instruction-Level Parallelism (ILP)?

- **To increase processor throughput by overlapped instruction execution**
  - **Used in most modern processors (even low-end ones)**
- **Two main approaches:**
  - **Hardware Techniques (dynamic): parallelism found and exploited by hardware during execution**
    - **Used in most modern processors (e.g. x86-64, ARM, RISC-V)**
  - **Software Techniques (static): parallelism found and exploited by compiler during compilation**
    - **Used in most modern compilers (at least the basic techniques)**
    - **Sophisticated schemes have limited use (in domain-specific scientific applications only)**
    - **e.g. Intel Itanium processors (~1999)**
- **In this lecture: we concentrate on basic software techniques to exploit ILP**

How to Exploit ILP?

- **ILP requires overlapped instruction execution**
- **A Basic Block: a block of consecutive instructions between two branch/jump instructions**
- **Average length of a basic block in RISC programs: ~6 instructions**
  - **Scope to exploit ILP is quite limited**
- **Observation: need to consider instructions across basic blocks!**
- **Simple example: ("loop-level parallelism")**
  - **Completely parallel loop**
  - **Each basic block is 7 instruction long (2 loads, 1 add, 1 store, 2 address updates, 1 branch)**
  - **Can be easily "unrolled" by compiler across loop iterations (explained later)**

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

Data Dependence (aka "True Dependence")

- **Instruction *j* is data dependent on instruction *i* if:**
  - **Instruction *i* produces a result that may be used by instruction *j***
  - **Instruction *j* is data dependent on instruction *k* and instruction *k* is data dependent on instruction *i***
- **Dependent instructions cannot be executed simultaneously or completely overlap**
- **Dependencies: property of programs**
  - **Pipeline organization determines if dependence is detected and if it causes a stall**
- **Data dependence conveys:**
  - **Possibility of a hazard**
  - **Order in which results must be calculated**
  - **Upper bound on exploitable instruction level parallelism**
- **Dependencies that flow through memory locations are difficult to detect**
  - **100(x4) and 20(x6) may be same memory location!**

Example: Data Dependence (aka "True Dependence")

```
Loop:  fld     f0,0(x1)     //f0=array element
       fadd.d  f4,f0,f2     //add scalar in f2
       fsd     f4,0(x1)     //store result
       addi    x1,x1,–8     //decrement pointer 8 bytes
       bne     x1,x2,Loop   //branch x1≠x2
```

Floating Point Data Dependence:

```
Loop:  fld     f0,0(x1)     //f0=array element
       fadd.d  f4,f0,f2     //add scalar in f2
       fsd     f4,0(x1)     //store result
```

Integer Data Dependence:

```
addi     x1,x1,-8   //decrement pointer
                    //8 bytes (per DW)
bne      x1,x2,Loop//branch x1ax2
```

Name Dependence

- ***Name***: a register or memory location
- Two instructions use the same *name* but no flow of information between them
- Suppose, instruction-*i* occurs before instruction-*j* in a program
- **Antidependence**: instruction *j* writes a register or memory location that instruction *i* reads
  - Order of register operation (*i* reading register before *j* updating it) must be preserved

```
fsd     f4,0(x1)     //store result
addi    x1,x1,-8     //decrement pointer 8 bytes
```

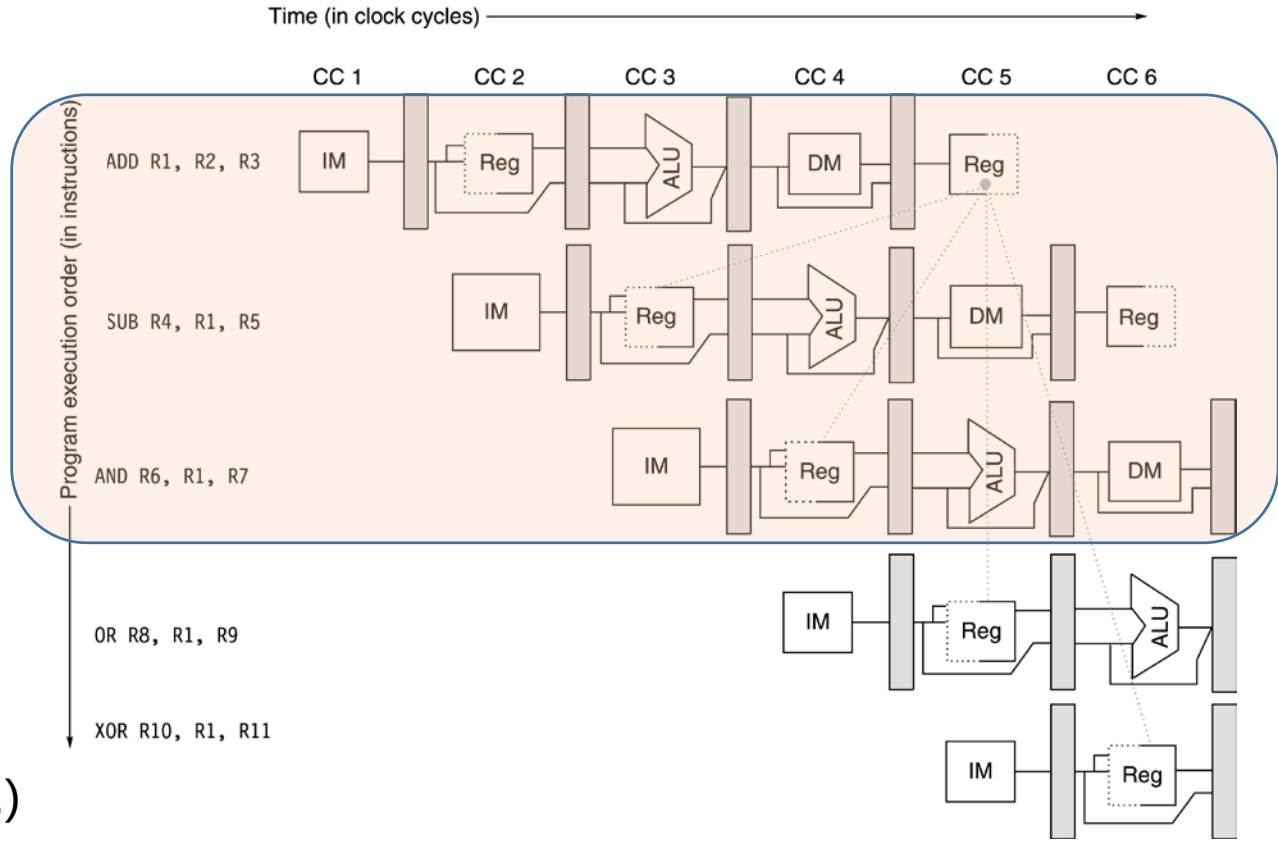
- **Output Dependence**: instruction *j* writes a register or memory location that instruction *i* also writes
  - Order of register operation (*i* reading register before *j* updating it) must be preserved

- Name dependence is usually solvable by changing the *name* in instructions

Recall: Data Hazards

- **Assume *i*-th instruction occurs before *j*-th instruction, both use register *x***
- **Read After Write (RAW) hazard:**
  - **Happens because of true dependence**
  - **Instr. *j* reads *x* before write by instr. *i* is complete, thereby using wrong value of *x***
  - **Can happen because WB is last pipeline stage!**
  - **Most common!**

Register *R1* causes RAW hazard between 1st and 2nd and 1st and 3rd instructions



OR: no hazard (why??)
XOR: no hazard, far away
(read happens after WB from ADD has completed)

H&P CA:A QA (6th. Ed.)

Recall: Data Hazards

- **Assume *i*-th instruction occurs before *j*-th instruction, both use register *x***
- **Write After Read (WAR) hazard:**
  - **Happens because of antidependence**
  - **Instr. *i* reads *x after* write by instr. *j*, thereby using wrong value of *x***
  - **Impossible in RISC-V pipeline (even for FP), because register value reading (during ID) is before register value update (during WB)**
  - **Can happen if instructions are executed out-of-order!**
- **Write After Write (WAW) hazard:**
  - **Happens because of true dependence**
  - **Instr. *i* writes *x after* write by instr. *j*, thereby setting wrong value of *x* going forward**
  - **Impossible in a simple 5-stage RISC pipeline**
  - **Again, can happen if instructions are executed out-of-order!**
- **Read after Read (RAR) is not a hazard!**

Control Dependence

- **Ensure *i*-th instruction is correctly ordered with respect to a branch instruction**
- **Instruction control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch**
  - **e.g. a statement within an if () {…} block cannot be moved before it!**

- **A processor can execute instructions that *should not have been executed,* if the correctness of the program is not affected!**
  - **e.g. the instruction in the *Branch Delay Slot***

- **Data and control dependence must to be preserved to ensure:**
  - **Data flow**
  - **Exception behavior**

Maintaining Data Flow and Exception Behavior

- **Data Flow:** ***actual*** **flow of data values between instructions that generate the values, and those who consume them**
  - ***Both*** **data dependence and control dependence must be maintained**

```
        add  x1,x2,x3
        beq  x4,x0,L
        sub  x1,x5,x6
L:      ...
        or   x7,x1,x8
```

or **instruction depends on *both* the** add **and** sub
but
value computed by or **depends on whether the** beq **succeeds!**

- **Exception Behavior: reordering of instructions should not cause any new exception in program!**

```
        add  x2,x3,x4
        beq  x2,x0,L1
        ld   x1,0(x2)
L1:
```

ld **instruction seems to be independent of** beq
Move ld **before** beq**?**
but
ld **can cause a memory access exception (e.g. a segmentation fault), which will not happen if** ld **remained after** beq **and** beq **succeeds!**

Sometimes Violating Control Dependence is OK…..

```
        add   x1,x2,x3
        beq   x12,x0,skip
        sub   x4,x5,x6
        add   x5,x4,x9
skip:   or    x7,x8,x9
```

Register x4 **is *never used* after the** or **instruction (“**x4 **is dead after the** or **instruction”) hence**
sub **can be moved before** beq **if it is guaranteed that** sub **will not generate exception!**
It’s execution might be useless if beq **succeeds, but program will be correct!**

Compiler Techniques for Exposing ILP

- **Basic idea: intelligent scheduling based on pipeline latency**
  - **Separate dependent instruction from the source instruction**
  - **Amount of separation: the pipeline latency of the source instruction**
- **Example:**

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

Assume the following pipeline 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

Scheduling to Decrease Stalls

f2: contains scalar value s
8(x1): pre-computed address of last element

```
Loop:  fld      f0,0(x1)
       stall
       fadd.d f4,f0,f2
       stall
       stall
       fsd f4,0(x1)
       addi x1,x1,-8
       bne x1,x2,Loop
```

8 clock cycles per iteration (3 stall cycles)

```
Loop:  fld      f0,0(x1)
       addi x1,x1,-8
       fadd.d f4,f0,f2
       stall
       stall
       fsd f4,0(x1)
       bne x1,x2,Loop
```

7 clock cycles per iteration (2 stall cycles)

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

Loop Unrolling to Decrease Stalls

- **Loop unrolling**
  - **Unroll by a factor of 4 (assume # elements is divisible by 4)**
  - **Eliminate unnecessary instructions**

1 cycle stall

2 cycles stall

```
Loop:  fld f0,0(x1)
       fadd.d f4,f0,f2
       fsd f4,0(x1) //drop addi & bne
       fld f6,-8(x1)
       fadd.d f8,f6,f2
       fsd f8,-8(x1) //drop addi & bne
       fld f0,-16(x1)
       fadd.d f12,f0,f2
       fsd f12,-16(x1) //drop addi & bne
       fld f14,-24(x1)
       fadd.d f16,f14,f2
       fsd f16,-24(x1)
       addi x1,x1,-32
       bne x1,x2,Loop
```

32(x1): pre-computed address of last block of 4 elements

12 stall clock cycles per iteration

26 clock cycles per 4 iterations

=> 6.5 clock cycles per iteration

Loop Unrolling with Scheduling: Decrease Stalls Further

- **Pipeline schedule the unrolled loop:**

```
Loop:  fld f0,0(x1)
       fld f6,-8(x1)
       fld f10,-16(x1)
       fld f14,-24(x1)
       fadd.d f4,f0,f2
       fadd.d f8,f6,f2
       fadd.d f12,f10,f2
       fadd.d f16,f14,f2
       fsd f4,0(x1)
       fsd f8,-8(x1)
       fsd f12,-16(x1)
       fsd f16,-24(x1)
       addi x1,x1,-32
       bne x1,x2,Loop
```

32(x1): pre-computed address of last block of 4 elements
0 stall clock cycles per iteration
14 clock cycles for 4 iterations
=> 3.5 clock cycles per iteration

Strip Mining

- Unknown number of loop iterations?
- Number of iterations = *n*
- Goal: make *k* copies of the loop body
- Generate pair of loops, one after the other:
  - First, executes a loop which runs for *n* mod *k* times
  - Then, execute *k*-fold unrolled loop that executes *n*/*k* times
    - Outer loop counter: runs *n*/*k* times
  - Technique is known as "Strip mining"

Thank you