COMP4611: Design and Analysis of Computer Architectures

Software Techniques and Compiler Support for Computer Architectures

LLP

Loop-Level Parallelism (LLP) Analysis

- Loop-Level Parallelism (LLP) analysis focuses on whether data accesses in later iterations of a loop are data dependent on data values produced in earlier iterations.

  e.g. in
  ```
  for (i=1; i<=1000; i++)
        x[i] = x[i] + s;
  ```

  The computation in each iteration is independent of the previous iterations and the loop is thus parallel. The use of x[i] twice is within a single iteration.

  $\Rightarrow$Thus loop iterations are independent from each other

- Loop-carried Dependence: A data dependence between different loop iterations (data produced in earlier iteration used in a later one) – limits parallelism.
- Instruction level parallelism (ILP) analysis, on the other hand, is usually done when instructions are generated by the compiler.

LLP Analysis Example 1

- In 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 */
}
```

(Where A, B, C are distinct non-overlapping arrays)

- **S2 uses the value A[i+1], computed by S1 in the same iteration. This data dependence is within the same iteration (not a loop-carried dependence).**

  ⇒ does not prevent loop iteration from being parallelized.

- **S1 uses a value computed by S1 in an earlier iteration, since iteration i computes A[i+1] read in iteration i+1 This is thus loop-carried dependence, and limits parallelism. The same applies for S2 for B[i] and B[i+1]**

  ⇒These two dependences are loop-carried spanning more than one iteration

LLP Analysis Example 2

- In 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  */
}
```

  - **S1 uses the value B[i] computed by S2 in the previous iteration (loop-carried dependence)**
  - **This dependence is not circular:**
    - S1 depends on S2 but S2 does not depend on S1.
  - **Can be made parallel by replacing the code with the following:**

```
A[1] = A[1] + B[1];              Loop Start-up code
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];        Loop Completion code
```

LLP Analysis Example 2

Original Loop:

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

Iteration 1
A[1] = A[1] + B[1];
B[2] = C[1] + D[1];

Iteration 2
A[2] = A[2] + B[2];
B[3] = C[2] + D[2];

. . . .
. . . .

Loop-carried Dependence

Iteration 99
A[99] = A[99] + B[99];
B[100] = C[99] + D[99];

Iteration 100
A[100] = A[100] + B[100];
B[101] = C[100] + D[100];

Modified Parallel Loop:

```
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];
```

Loop Start-up code
A[1] = A[1] + B[1];
B[2] = C[1] + D[1];

Iteration 1
A[2] = A[2] + B[2];
B[3] = C[2] + D[2];

. . . .

Not Loop Carried Dependence

Iteration 98
A[99] = A[99] + B[99];
B[100] = C[99] + D[99];

Iteration 99
A[100] = A[100] + B[100];
B[101] = C[100] + D[100];

Loop Completion code

The Role of Compilers

Compiler and ISA

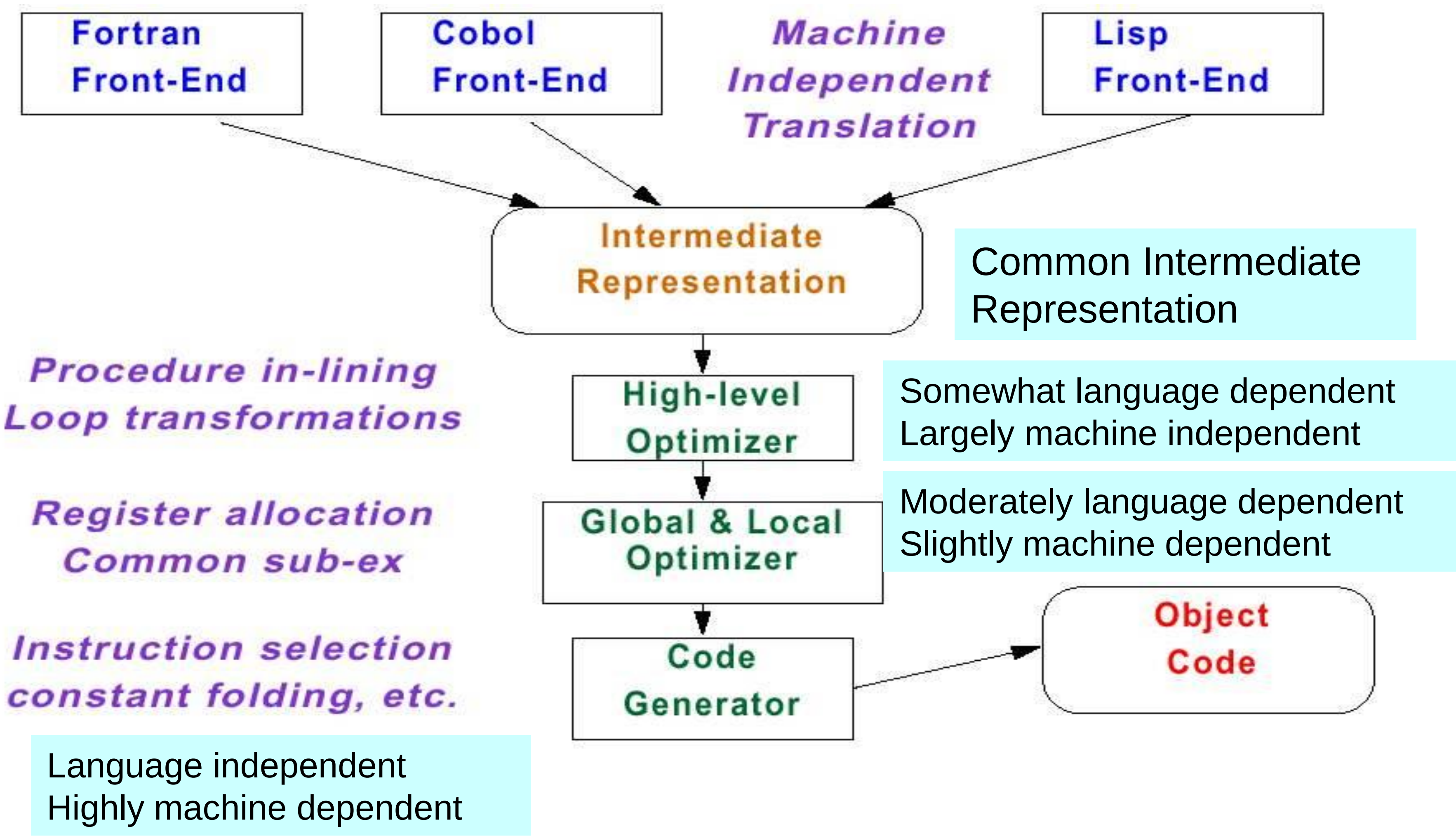
- ISA decisions are no longer just for programming in assembly languages (AL) easily
- With HLLs, ISA is a compiler target today
- Performance of a computer will be significantly affected by compilers
- Understanding the compiler technology today is critical to designing and efficiently implementing an instruction set
- Architectural choices affect the code quality and the complexity of building a compiler for it

Goal of the Compiler

- Primary goal is correctness
- Second goal is speed of the object code
- Others:
  - Speed of the compilation
  - Ease of providing debug support
  - Inter-operability among languages
  - Flexibility of the implementation - languages may not change much but they do evolve - e. g. Fortran 66 ===> HPF

Make the common cases fast and the rare cases correct

Typical Modern Compiler Structure



Optimization Types

- High level - done at source code level
  - E.g., procedure called only once - so put it in-line and save CALL
- Local - done on a basic block (straight-line code)
  - Common sub-expressions produce same value
  - Constant propagation - replace constant valued variable with the constant - saves multiple variable accesses with same value
- Global - same as local but done across branches
  - Code motion - remove code from loops that compute same value on each pass and put it before the loop
  - Simplify or eliminate array addressing calculations in loop

Optimization Types (Cont.)

- Register allocation
  - Use graph coloring (graph theory) to allocate registers
    - NP-complete
    - Heuristic algorithm works best when there are at least 16 (and preferably more) registers
- Processor-dependent optimization
  - Strength reduction: replace multiply with shift and add sequence
  - Pipeline scheduling: reorder instructions to minimize pipeline stalls
  - Branch offset optimization: Reorder code to minimize branch offsets

Strength reduction

Example:

```
for (j = 0; j = n; ++j)
   A[j] = 2*j;

for (i = 0; 4*i <= n; ++i)
   A[4*i] = 0;
```

An optimizing compiler can replace multiplication by 4 by addition of 4.

Constant propagation

```
a = 5;
 ...
// no change to a so far.
if (a > b)
    {
    . . .
    }
```

The statement `(a > b)` can be replaced by `(5 > b)`. This could free a register when the comparison is executed.

When applied systematically, constant propagation can improve the code significantly.

Register Allocation

- One the most important optimizations
- Based on graph coloring techniques
  - Construct graph based on the liveness of registers
    - Use a vertex to represent a variable,
    - Add an edge between two vertices if the two corresponding variables are live at the same time
  - If there are k registers, use k-coloring to allocate registers
  - Goal is to achieve 100% register allocation for all active variables.
  - Graph coloring works best when there are at least 16 general-purpose registers available for integers and more for floating-point variables.

Major Types of Optimizations and Examples

Optimization name	Explanation	Percentage of the total number of optimizing transforms
High-level	*At or near the source level; processor-independent*	
Procedure integration	Replace procedure call by procedure body	N.M.
Local	*Within straight-line code*	
Common subexpression elimination	Replace two instances of the same computation by single copy	18%
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	22%
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation	N.M.
Global	*Across a branch*	
Global common subexpression elimination	Same as local, but this version crosses branches	13%
Copy propagation	Replace all instances of a variable A that has been assigned X (i.e., $A = X$) with X	11%
Code motion	Remove code from a loop that computes same value each iteration of the loop	16%
Induction variable elimination	Simplify/eliminate array addressing calculations within loops	2%
Processor-dependent	*Depends on processor knowledge*	
Strength reduction	Many examples, such as replace multiply by a constant with adds and shifts	N.M.
Pipeline scheduling	Reorder instructions to improve pipeline performance	N.M.
Branch offset optimization	Choose the shortest branch displacement that reaches target	N.M.

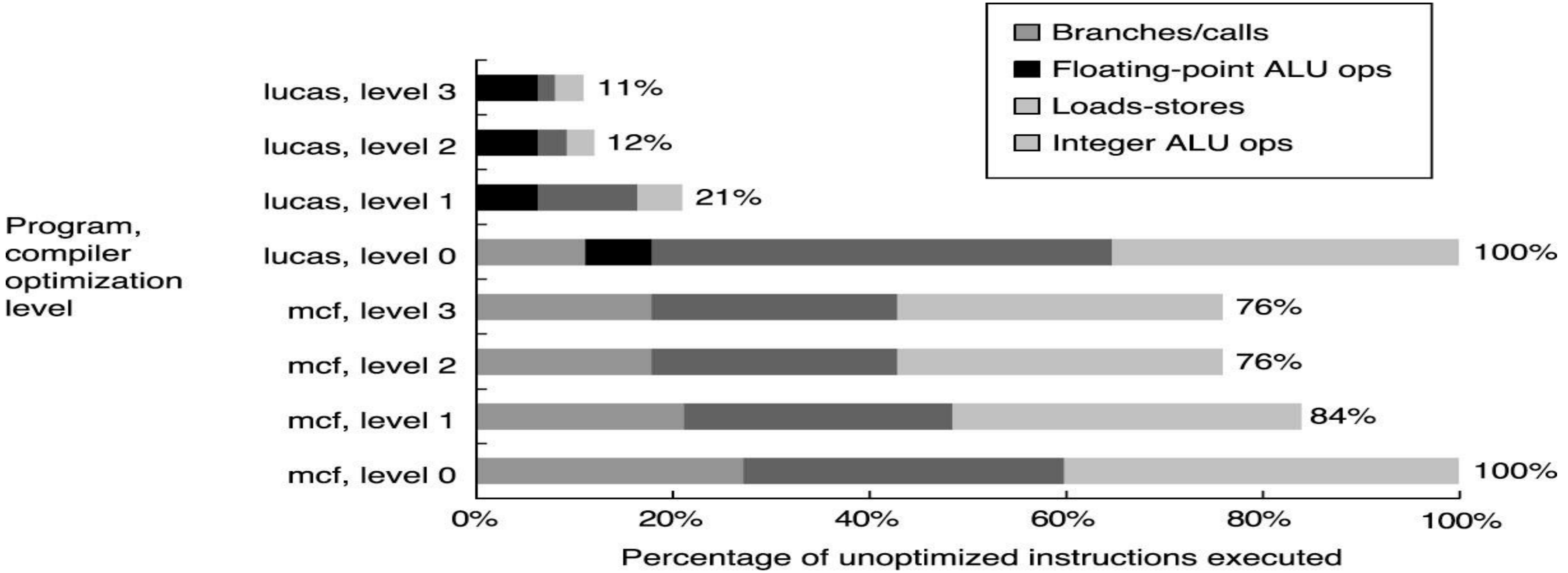
Practice Makes Perfection

- gcc optimization flags “-O1, -O2, -O3”
  - “-O0” turns off optimization
  - Example: gcc –O3 –o <out_file> <in_file>

- Examine the binary
  - objdump –D <executable or obj file>
  - View the output: “less <file>”

- Write a program, and see the difference with different optimization flags

Change in IC Due to Optimization



- Level 1: local optimizations, code scheduling, and local register allocation
- Level 2: global optimization, loop transformation (software pipelining), global register allocation
- Level 3: + procedure integration

How can Architects Help Compiler Writers

- Provide Regularity
  - Addressing modes, operations, and data types should be orthogonal (independent) of each other
    - Simplify code generation especially multi-pass
    - Counterexample: restrict what registers can be used for a certain class of instructions
- Provide primitives - not solutions
  - Special features that match an HLL construct are often un-usable
  - What works in one language may be detrimental to others

How can Architects Help Compiler Writers (Cont.)

- Simplify trade-offs among alternatives
  - How to write good code? What is a good code?
    - Metric: IC or code size (no longer true) →caches and pipeline…
  - Anything that makes code's performance easier to estimate
    - How many times a variable should be referenced before it is cheaper to load it into a register
- Provide instructions that bind the quantities known at compile time as constants
  - Don't hide compile time constants
    - Instructions which work off of something that the compiler thinks could be a run-time determined value hand-cuff the optimizer

Short Summary -- Compilers

- ISA has at least 16 GPRs (not counting FP registers) to simplify allocation of registers using graph coloring
- Orthogonality suggests all supported addressing modes apply to all instructions that transfer data
- Simplicity – understand that less is more in ISA design
  - Provide primitives instead of solutions
  - Simplify trade-offs between alternatives
  - Don’t bind constants at runtime

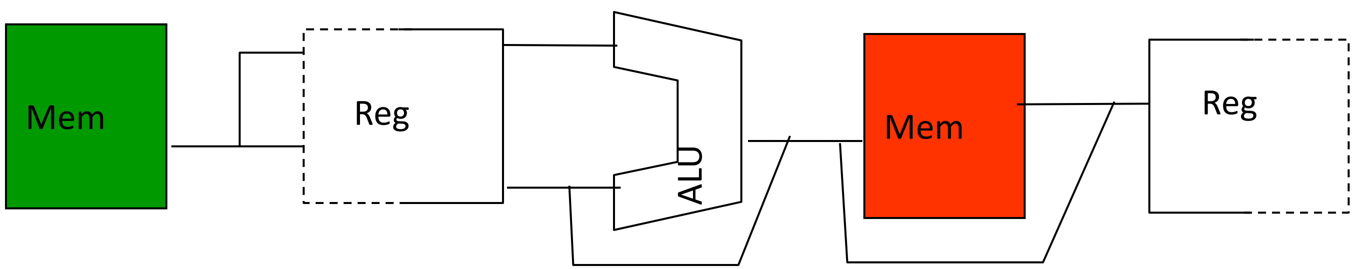
A Summary of Pipelining

Recall from Pipelining

- Pipeline CPI = Ideal pipeline CPI + Structural Stalls + Data Hazard Stalls + Control Stalls
  - Ideal pipeline CPI: measure of the maximum performance attainable by the implementation
  - Structural hazards: HW cannot support this combination of instructions
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps)

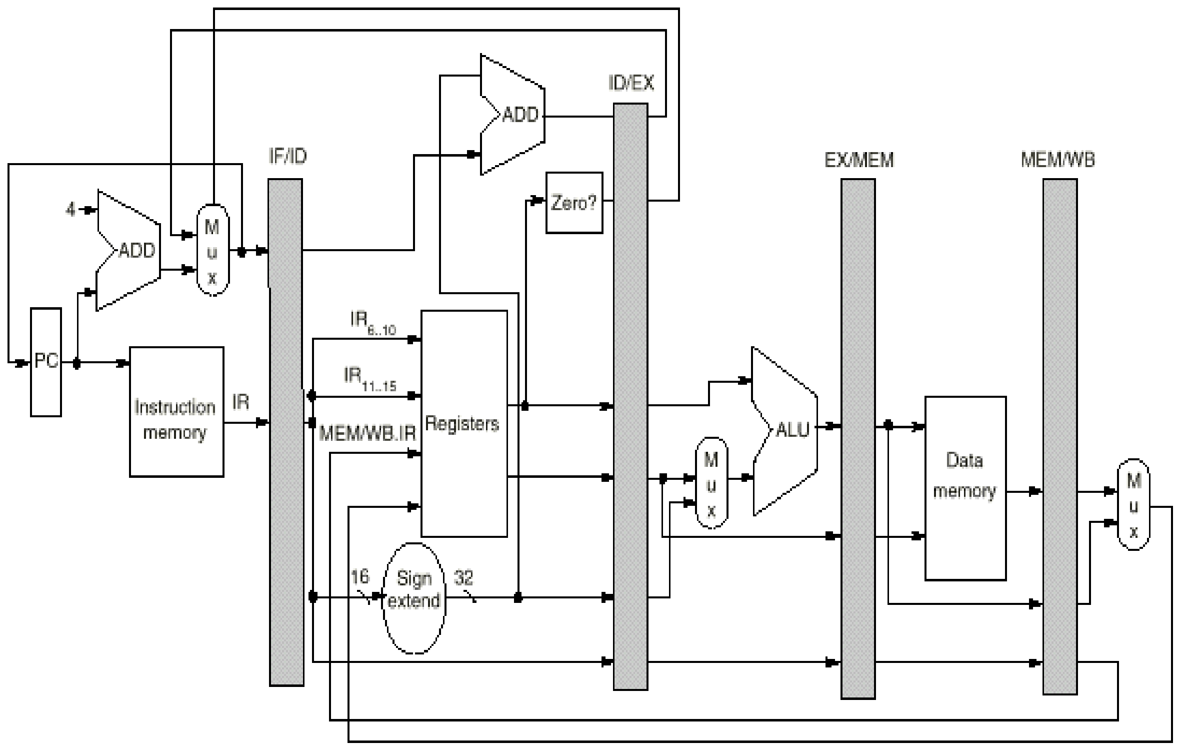
Techniques to Reduce Stalls and Increase ILP

- **Hardware** Schemes to Reduce:
  - Structural hazards
    - ✓ Memory: Separate instruction and data memory
    - ✓ Registers: Write 1st half of cycle and read 2nd half of cycle



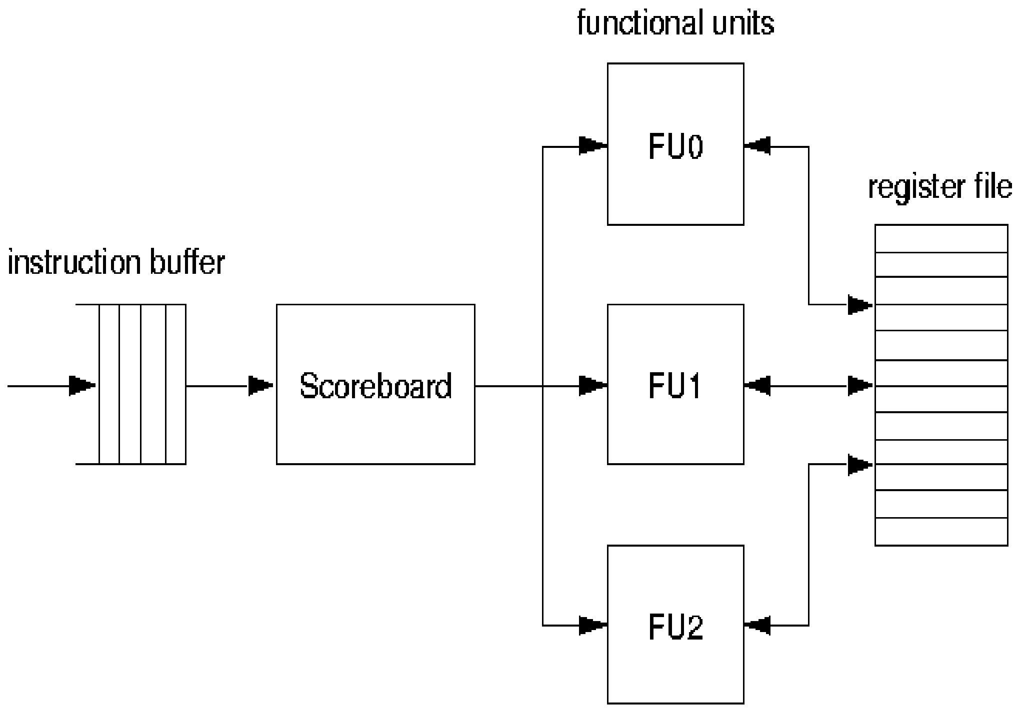
Techniques to Reduce Stalls

- **Hardware** Schemes to Reduce:
  - ❑ Data Hazards
    - ✓ Forwarding
  - ❑ Control Hazards
    - ✓ Moving the branch resolution earlier in the pipeline



Techniques to Reduce Stalls and Increase ILP

- **Hardware** Schemes to increase ILP:
  - ❑ Scoreboarding
    - ✓ Allows out-of-order execution of instructions



Techniques to Reduce Stalls and Increase ILP

- **Hardware** Schemes to increase ILP:
  - ❑ Scoreboarding
    - ✓ Allows out-of-order execution of instructions

Instruction status					Read	Execution	Write
Instruction	j	k		Issue	operands	complete	Result
L.D	F6	34+	R2	1	2	3	4
L.D	F2	45+	R3	5	6	7	8
MUL.D	F0	F2	F4	6	9	19	20
SUB.D	F8	F6	F2	7	9	11	12
DIV.D	F10	F0	F6	8	21	61	62
ADD.D	F6	F8	F2	13	14	16	22

- We have:
  - In-oder issue,
  - Out-of-order execute and "completion"

Techniques to Reduce Stalls and Increase ILP

- **Hardware** Schemes to reduce stalls
  - The Tomasulo's Algorithm
    - ✓ Similar to scoreboarding but more advanced (e.g., register renaming)
  - Control Hazards
    - ✓ Dynamic branch prediction (using buffer lookup schemes)

Techniques to Reduce Stalls and Increase ILP

- **Software** Schemes to Reduce:
  - Data Hazards
    - ✓ Compiler Scheduling: reduce load stalls

Original code with stalls:

	LD	Rb,b
Stall →	LD	Rc,c
	DADD	Ra,Rb,Rc
	SD	Ra,a
	LD	Re,e
Stall →	LD	Rf,f
	DSUB	Rd,Re,Rf
	SD	Rd,d

Scheduled code with no stalls:

LD	Rb,b
LD	Rc,c
LD	Re,e
DADD	Ra,Rb,Rc
LD	Rf,f
SD	Ra,a
DSUB	Rd,Re,Rf
SD	Rd,d

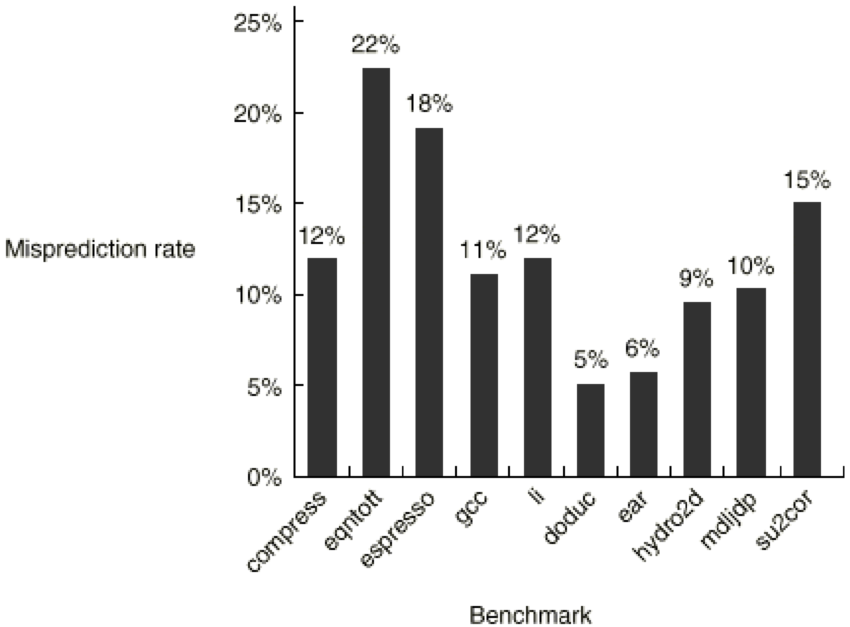
Techniques to Reduce Stalls and Increase ILP

- **Software** Schemes to Reduce:
  - Data Hazards
    - ✓ Compiler Scheduling: register renaming to eliminate WAW and WAR hazards

MUL.D	F0, F1, F2	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
ADD.D	F0, F3, F4		IF	ID	A1	A2	A3	A4	MEM	WB		

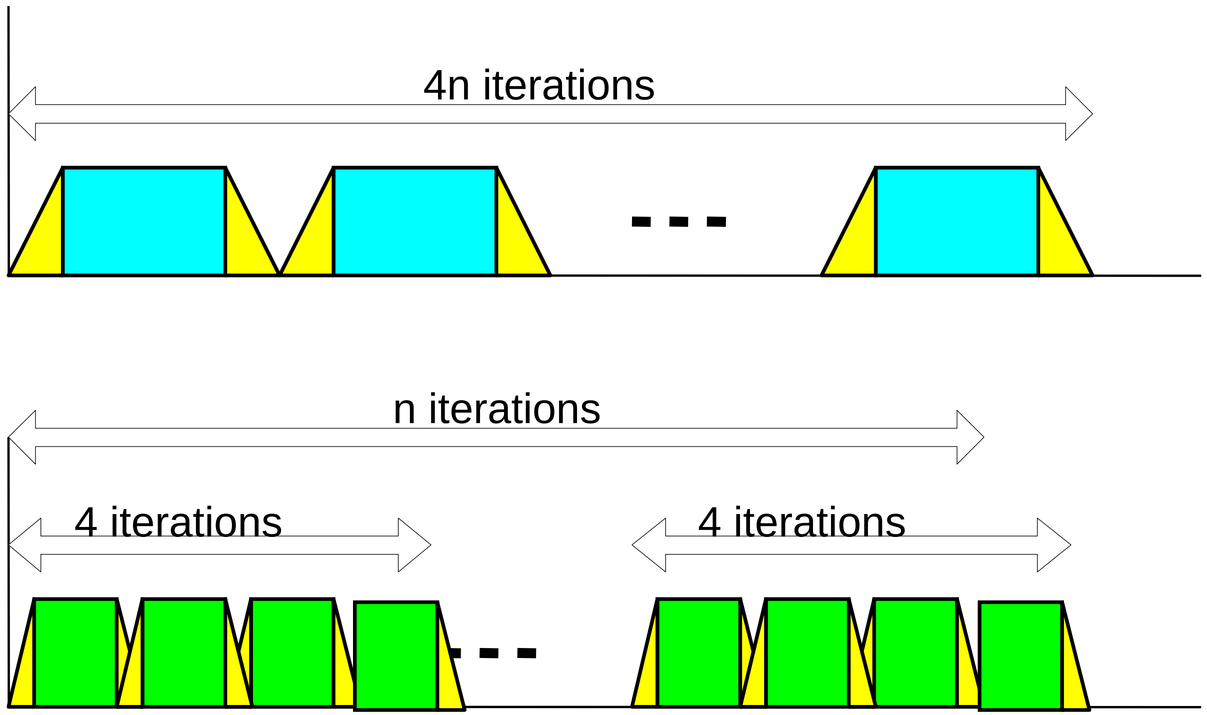
Techniques to Reduce Stalls and Increase ILP

- **Software** Schemes to Reduce:
  - ❑ Control Hazards
    - ✓ Branch prediction
      - ✓ Example : choosing backward branches **(loop)** as taken and forward branches **(if)** as not taken
      - ✓ Tracing Program behaviour



Techniques to Reduce Stalls and Increase ILP

- **Software** Schemes to Reduce:
  - ❑ Control Hazards
    - ✓ Loop unrolling



Techniques to Reduce Stalls and Increase ILP

- **Software** Schemes to Reduce:
  - ❑ Control Hazards
    - ✓ Increase loop-level parallelism

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

– Can be made parallel by replacing the code with the following:

```
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];
```