Instruction Level Parallelism

Jyoti Kumari

Instruction Level Parallelism (ILP)

- All processors use pipelining to overlap the execution of instructions and improve performance.
- This potential overlap among instructions is called *instruction-level parallelism* (ILP), since the instructions can be evaluated in parallel.
- ILP is mostly exploited in the use of branches.
- A “basic block” is a block of code that has no branches into or out of except for at the start and the end.
- The amount of parallelism available within a basic block is quite small. The average dynamic branch frequency in integer programs was measured to be about 15%, meaning that about 7 instructions execute between a pair of branches.
- Since the instructions are likely to depend upon one another, the amount of overlap we can exploit within a basic block is likely to be much less than 7.
- To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks.

- The simplest and most common way to increase the amount of parallelism available among instructions is to exploit parallelism among iterations of a loop. This type of parallelism is often called ***loop-level parallelism***.
- Example : A loop that adds two 1000-element arrays and is completely parallel.

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

- Every iteration of the loop can overlap with any other iteration, although within each loop iteration there is little or no opportunity for overlap.
- Techniques for converting such loop-level parallelism into instruction-level parallelism.
  - unrolling the loop either statically by the compiler
  - dynamically by the hardware

Example 2

```
for (i=1; i<=100; i= i+1){
  a[i] = a[i] + b[i];        //s1
  b[i+1] = c[i] + d[i];      //s2
}
```

Is this loop parallel? If not how to make it parallel?

- Statement **s1** uses the value assigned in the previous iteration by statement **s2**, so there is a loop-carried dependency between **s1** and **s2**.
- Despite this dependency, this loop can be made parallel because the dependency is not circular:
  - neither statement depends on itself;
  - while **s1** depends on **s2**, **s2** does not depend on **s1**.

- A loop is parallel unless there is a cycle in the dependecies, since the absence of a cycle means that the dependencies give a partial ordering on the statements.

- To expose the parallelism the loop must be transformed to conform to the partial order. Two observations are critical to this transformation:
  - There is no dependency from **s1** to **s2**. Then, interchanging the two statements will not affect the execution of **s2**.
  - On the first iteration of the loop, statement **s1** depends on the value of **b[1]** computed prior to initiating the loop.
- This allows us to replace the loop above with the following code sequence, which makes possible overlapping of the iterations of the 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];
```

Example 3

```
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}
```

- This loop is not parallel because it has cycles in the dependencies, namely the statements **S1** and **S2** depend on themselves!

Dependences and Hazards

1. **Data Dependence:**
   - Instruction *i* produces a result that may be used by instruction *j*
   - instruction *j* is data dependent on instruction *k* and vice versa.
2. **Name Dependence:**
   - Occurs when two instructions use the same register and memory location, called a name. But there is no flow of data between the instructions associated with the name.
   - There are two types of name dependences between an instruction *i* that *precedes* instruction *j* in program order:
     - Antidependence: *j* writes to a location that *i* reads.
     - Output Dependence: two instructions write to the same location.
1. **Control Dependence: Discussed later**

Dependences and Hazards: Types of Data Hazards:

Consider two instructions *i* and *j*, with *i* preceding *j* in program order. The possible data hazards are

1. RAW *(read after write)*—*j* tries to read a source before *i* writes it, so *j* incorrectly gets the *old* value.
   - This hazard is the most common type and corresponds to a true data dependence.
   - Program order must be preserved to ensure that *j* receives the value from *i*.
2. WAW *(write after write)*—*j* tries to write an operand before it is written by *i*.
   - The writes end up being performed in the wrong order, leaving the value written by *i* rather than the value written by *j* in the destination.
   - This hazard corresponds to an output dependence.
   - WAW hazards are present only in pipelines that write in more than one pipe stage or allow an instruction to proceed even when a previous instruction is stalled.

3. WAR *(write after read)*—*j* tries to write a destination before it is read by *i*, so *i* incorrectly gets the *new* value.
   - This hazard arises from an antidependence (or name dependence).
   - WAR hazards cannot occur in most static issue pipelines— even deeper pipelines or floating-point pipelines—because all reads are early (ID) and all writes are late (WB).
   - A WAR hazard occurs either when there are some instructions that write results early in the instruction pipeline *and* other instructions that read a source late in the pipeline, or when instructions are reordered.
4. RAR *(read after read)* case is not a hazard.

Control Dependence

- A control dependence determines the ordering of an instruction, *i*, with respect to a branch instruction so that instruction *i* is executed in correct program order and only when it should be.

- Assume we have the following piece of code:

```
If p1{
S1;
}

If p2{
S2;
}
```

- S1 is dependent on p1 and S2 is dependent on p2 but not on p1.

Control Dependence

- Control Dependences have the following constraints:
  - An instruction that is control dependent on a branch cannot be moved before the branch, so that the branch no longer controls it.
  - An instruction that is not control dependent on a branch cannot be moved after the branch so that the branch controls it.

Techniques of overcoming data hazards

- Static Scheduling- It is the instruction scheduling specified by the compiler in the machine code it generates. E.g., Loop Unrolling
- Dynamic Scheduling: The processor control logic schedules instruction on the fly taking into account inter-instruction dependences as well as the state of the functional units.

Static Scheduling-Loop Unrolling

- To keep a pipeline full, parallelism among instructions must be exploited by finding sequences of unrelated instructions that can be overlapped in the pipeline.
- To avoid stalls, a dependent instruction must be separated from the source instruction by a distance in clock cycles equal to the pipeline latency of that source instruction.
- A compiler's ability to perform this scheduling depends both on the amount of ILP available in the program and on the latencies of the functional units in the pipeline. Figure 3.1 shows the (Floating Point) FP unit 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

Figure 3.1

Independent instructions from multiple successive iterations of a loop can be made to execute in parallel.

Example: A simple loop that adds a scalar value to an array in memory:

```
for (i=1; i<=1000; i++)
  x[i] = x[i] + s;
```

1. The first step is to translate the above segment to MIPS assembly language.

Loop:	LD	F0, 0(R1)	;F0 - array element
	ADDD	F4, F0, F2	;add scalar in F2
	SD	0(R1), F4	;store result
	SUBI	R1, R1, #8	;decrement pointer ;8 bytes (per double)
	BENZ	R1, Loop	;branch R1 != zero

We can schedule the loop to obtain only two stalls and reduce the time to 6 cycles.

2. Show how the loop would look on MIPS, both scheduled and unscheduled, including any stalls or idle clock cycles. Schedule for both delays
   - from floating-point operations and
   - from the delayed branches.

Without any scheduling		Scheduled		
	Cycles		Cycles	
Loop: LD F0, 0(R1)	1	Loop: LD F0, 0(R1)	1	
stall	2	stall	2	
ADDD F4, F0,F2	3	ADDD F4, F0, F2	3	
stall	4	SUBI R1, R1, #8	4	
stall	5	BENZ R1,Loop	5	;delayed branch
SD 0(R1), F4	6	SD 8(R1), F4	6	;altered and interchanged with SUBI
SUBI R1, R1,#8	7			
BENZ R1, Loop	8			
stall	9			
9 clock cycles per element		6 clock cycles per element		

In the previous example, we complete one loop iteration and store back one array element every 6 clock cycles, but the actual work of operating on the array element takes just three (the load, add, and store) of those 6 clock cycles.

- The remaining four clock cycles consist of loop overhead—the SUBI and BNE—and two stalls.
- To eliminate these four clock cycles we need to get more operations relative to the number of overhead instructions.
- A simple scheme for increasing the number of instructions relative to the branch and overhead instructions is *loop unrolling*.
- Unrolling simply replicates the loop body multiple times, adjusting the loop termination code.
- Loop unrolling can also be used to improve scheduling. Because it eliminates the branch, it allows instructions from different iterations to be scheduled together.
- In this case, we can eliminate the data use stalls by creating additional independent instructions within the loop body.
- If we simply replicated the instructions when we unrolled the loop, the resulting use of the same registers could prevent us from effectively scheduling the loop.
- Thus, we will want to use different registers for each iteration, increasing the required number of registers.

3. Show the loop unrolled (scheduled and unscheduled) so that there are 4 copies of the loop body, assuming R1 is initially a multiple of 32, which means that the number of loop iterations is a multiple of 4. Eliminate any obviously redundant computations, and do not reuse any of the registers.

Without any scheduling

```
Loop: LD    F0, 0(R1)          1
      stall                    2
      ADDD  F4, F0, F2         3
      stall                    4
      stall                    5
      SD    0(R1), F4          6   ;drop SUBI &BNEZ
      LD    F6, -8(R1)         7
      stall                    8
      ADDD  F8, F6, F2         9
      stall                    10
      stall                    11
      SD    -8(R1), F8         12  ;drop SUBI &BNEZ
      LD    F10,-16(R1)        13
      stall                    14
      ADDD  F12,F10,F2         15
      stall                    16
      stall                    17
      SD    -16(R1), F12       18  ;drop SUBI &BNEZ
      LD    F14,-24(R1)        19
      stall                    20
      ADDD  F16,F14,F2         21
      stall                    22
      stall                    23
      SD    -24(R1),F16        24
      SUBI  R1, R1, #32        25
      BENZ  R1, Loop           26
      stall                    27
```

27 clock cycles per iteration;
27/4 = **6.8 clock cycles per element**

Scheduled

```
Loop: LD    F0, 0(R1)          1
      LD    F6, -8(R1)         2
      LD    F10,-16(R1)        3
      LD    F14,-24(R1)        4
      ADDD  F4, F0, F2         5
      ADDD  F8, F6, F2         6
      ADDD  F8, F6, F2         7
      ADDD  F16, F14, F2       8
      SD    0(R1), F4          9
      SD    -8(R1), F8         10
      SD    -16(R1), F12       11
      SUBI  R1, R1, #32        12
      BENZ  R1, Loop           13
      SD    8(R1), F16         14  ;8-32=-24
```

14 clock cycles per iteration;
14/4 = **3.5 clock cycles per element**

Dynamic Scheduling (continued)

- A major limitation of the pipelining techniques is that they use in-order instruction issue: ***if an instruction is stalled in the pipeline, no later instructions can proceed***.
- Thus, data dependence can cause stalling in a pipeline that has "long" execution times for instructions that dependencies.
- EX: Consider this code (.D is floating point) ,

```
DIV.D F0,F2,F4
ADD.D F10,F0,F8
SUB.D F12,F8,F14
```

- The SUB.D instruction cannot execute because the dependence of ADD.D on DIV.D causes the pipeline to stall; yet, SUB.D is not data dependent on anything in the pipeline.
- This hazard creates a performance limitation that can be eliminated by not requiring instructions to execute in program order.

Summary

- Loop unrolling is normally done early in the compilation process, so that redundant computations can be exposed and eliminated by the optimizer.
- To obtain the final unrolled code we had to make the following decisions and transformations:
  - Determine that unrolling the loop would be useful by finding that the loop iterations were independent, except for the loop maintenance code.
  - Use different registers to avoid unnecessary constraints that would be forced by using the same registers for different computations (e.g., name dependences).
  - Eliminate the extra test and branch instructions and adjust the loop termination and iteration code.
  - Determine that the loads and stores in the unrolled loop can be interchanged by observing that the loads and stores from different iterations are independent.
  - This transformation requires analyzing the memory addresses and finding that they do not refer to the same address.
  - Schedule the code, preserving any dependences needed to yield the same result as the original code.

Dynamic Scheduling

- The previous example that we looked at was an example of *statically scheduled* pipeline.
- Instructions are fetched and then issued. If the users code has a data dependency / control dependence it is hidden by forwarding.
- If the dependence cannot be hidden a stall occurs.
- *Dynamic Scheduling* is an important technique in which both dataflow and exception behavior of the program are maintained.
- Here, the hardware rearranges the instruction execution to reduce the stalls.

- Dynamic scheduling offers several ***advantages***:
- It enables handling some cases when dependencies are unknown at compile time (e.g., because they may involve a memory reference);
- It simplifies the compiler;
- It allows code that was compiled with one pipeline in mind to run efficiently on a different pipeline.

- To allow us to begin executing the SUB.D in the above example, we must separate the issue process into two parts:
  - checking for any structural hazards and
  - waiting for the absence of a data hazard.
- Thus, we still use in-order instruction issue (i.e., instructions issued in program order), but we want an instruction to begin execution as soon as its data operands are available.
- Such a pipeline does *out-of-order execution*, which implies *out-of-order completion*.

Dynamic Scheduling (continued)

- Longer execution time of certain floating point numbers leads to WAR and WAW hazards.
- Anti-dependence between the ADD.D and the SUB.D
- if the pipeline executes the SUB.D before the ADD.D (which is waiting for the DIV.D), it will violate the anti-dependence, yielding a WAR hazard.
- To avoid violating output dependences, such as the write of F6 by MUL.D, WAW hazards must be handled.

Dynamic Scheduling (continued)

- If we want to execute instructions out of order in hardware (if they are not dependent etc.) we need to modify the ID stage of our 5 stage pipeline.
- Split ID into the following stages:
  - *Issue*: Decode instructions, check for structural hazards.
  - *Read Operands*: Wait until no data hazards, then read operands.
- IF still precedes ID and will store the instruction into a register or queue.

Dynamic Scheduling: Score-boarding

- It is a technique for allowing instructions to execute out of order when there are sufficient resources and no data dependences.
- It is named after the CDC 6600 scoreboard, which developed this capability.

Dynamic Scheduling: Tomasulo's Approach

- Tomasulo's Algorithim was invented by Robert Tomasulo.
- It was used in the IBM 360/391 floating-point unit to allow out-of-order execution.
- The algorithm will avoid RAW hazards by executing an instruction only when it's operands are available. WAR and WAW hazards are avoided by register renaming.

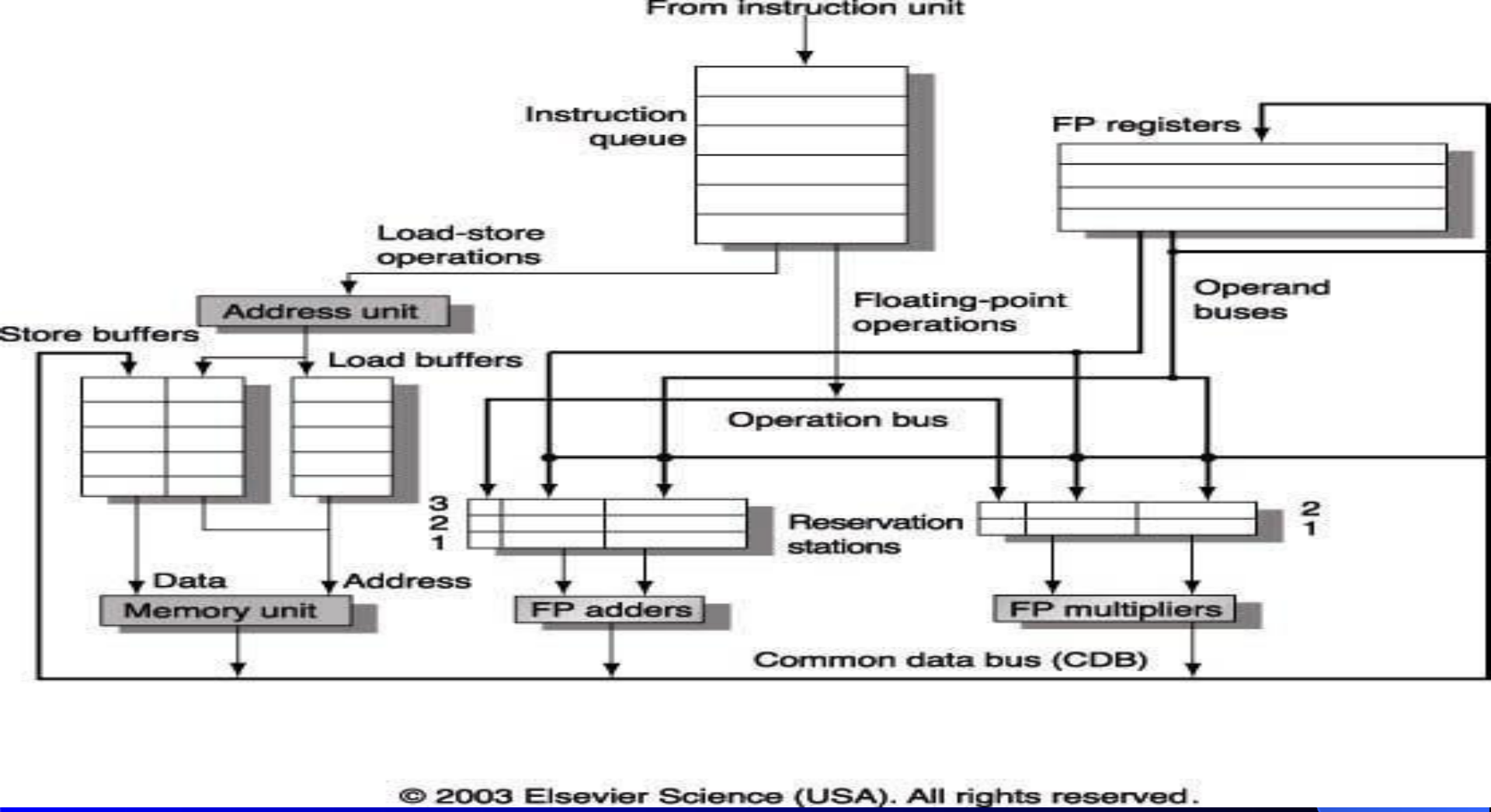
```
DIV.D F0,F2,F4
ADD.D F6,F0,F8
S.D   F6,0(R1)
SUB.D F8,F10,F14
MUL.D F6,F10,F8
```

```
DIV.D F0,F2,F4
ADD.D Temp,F0,F8
S.D   Temp,0(R1)
SUB.D Temp2,F10,F14
MUL.D F6,F10,Temp2
```

- Anti-dependence: Output Dependence
- 1. ADD.D & SUB.D ADD.D & MUL.D
- 2. S.D & MUL.D

- Leading to
- WAR hazards on the use of F8 by ADD.D and the use of F6 by the SUB.D.
- A WAW hazard since the ADD.D may finish later than the MUL.D.
- There are also three true data dependences: between the DIV.D and the ADD.D, between the SUB.D and the MUL.D, and between the ADD.D and the S.D.

- In Tomasulo's scheme, register renaming is provided by *reservation stations*, which buffer the operands of instructions waiting to issue.
- The basic idea is that a reservation station fetches and buffers an operand as soon as it is available, eliminating the need to get the operand from a register.
- In addition, pending instructions designate the reservation station that will provide their input.
- Finally, when successive writes to a register overlap in execution, only the last one is actually used to update the register.
- As instructions are issued, the register specifiers for pending operands are renamed to the names of the reservation station, which provides register renaming.



The basic structure of a MIPS floating-point unit using Tomasulo's algorithm.

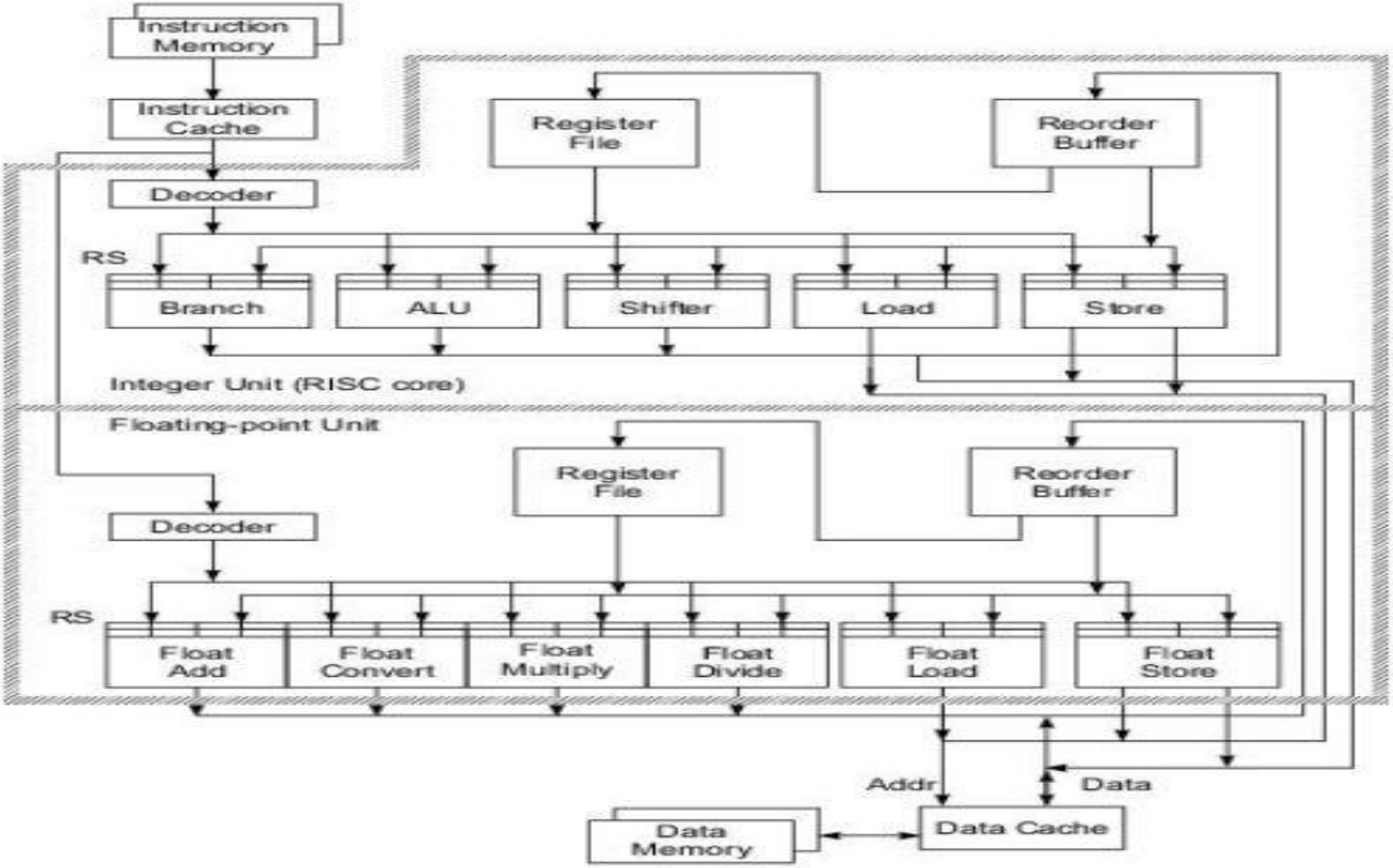
Advanced Processor Technology

- Superscalar Processor- Subclass of RISC processors- which allow multiple instructions to be issued simultaneously during each cycle.
- VLIW Processor (Very long instruction word)- use more functional units than a superscalar processor.
- Super-pipelined Processor
- Vector Processor
- Symbolic Processors

- Scalar and vector processors are for numerical computations.
- Symbolic processors have been developed for AI applications.

Superscalar Processors

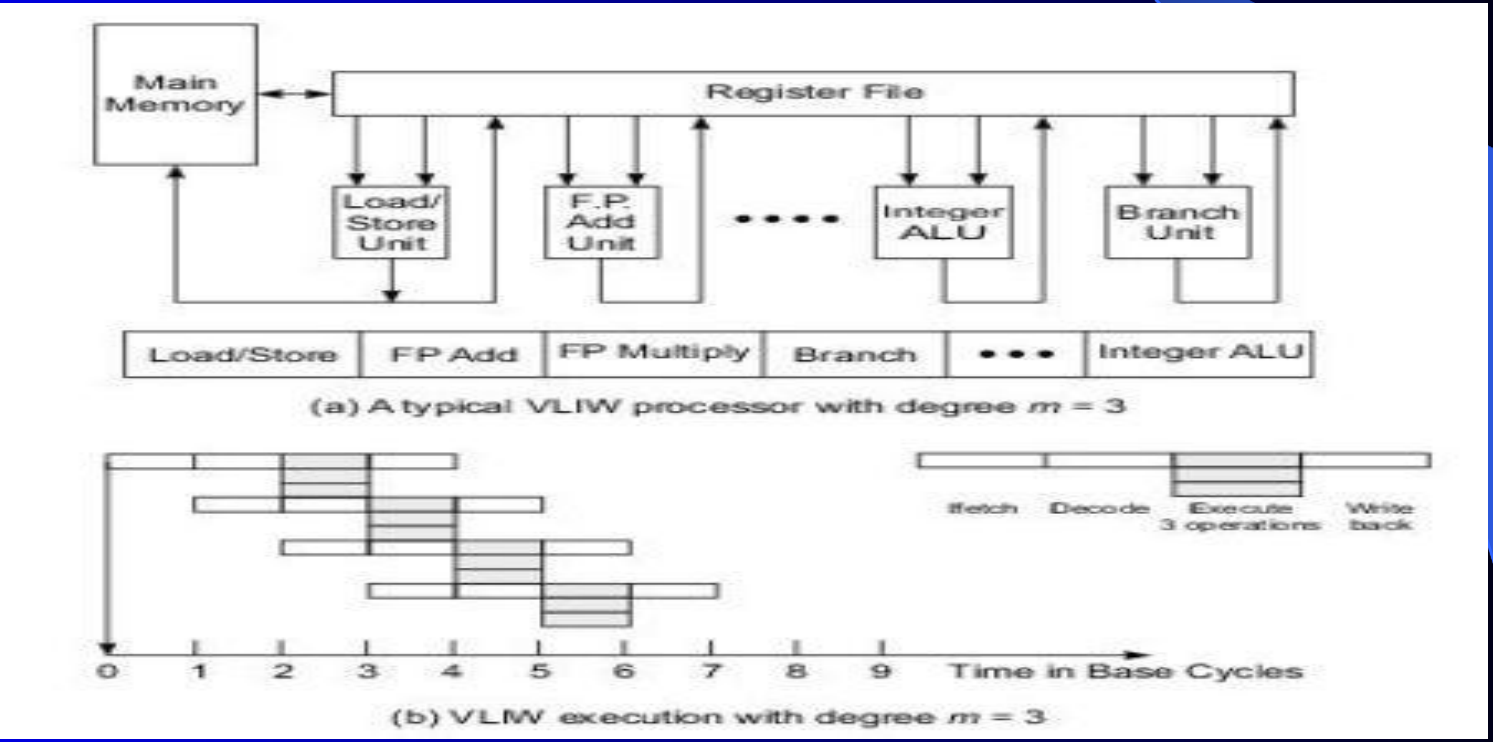
- ACISC or a RISC scalar processor can be improved with a superscalar or vector architecture.
  - Scalar processors are those executing one instruction per cycle.
  - Only one instruction is issued per cycle, and only one completion of instruction is expected from the pipeline per cycle.
- In a superscalar processor, multiple instructions are issued per cycle and multiple results are generated per cycle.
- A vector processor executes vector instructions on arrays of data;
- Superscalar Processors are designed to exploit more instruction-level parallelism in user programs.



A typical superscalar RISC processor architecture consisting of an integer unit and floating point unit.

VLIW Architecture

- The VLIW architecture is generalized from two well-established concepts: horizontal microcoding and superscalar processing.
- A typical VLIW machine has instruction words hundreds of bits in length.



Vector and Symbolic Processors

- Designed to perform vector computations.
- A vector processor can assume either a register-to-register architecture or a memory-to-memory
  architecture.
- The former uses shorter instructions and vector register files.
- The latter uses memory-based instructions which are longer in length, including memory addresses.
-

Vector Instructions

- Register-based vector instructions appear in most register-to-register vector processors like Cray supercomputers.
- V_i = Vector register of length n
- s_i = Scalar Register
- M(1:n) = Array of length n

- Register-based vector operations
- V_1 o V_2 - V_3 (binary vector)
- s_1 o V_1 - V_2 (scaling)
- V_1 o V_2 - s_1 (binary reduction)
- M(1:n) - V_1 (vector load)
- V_1 - M(1:n) (vector store)
- o V_1 - V_2 (unary vector)
- o V_1 - s_1 (unary Vector)