Advanced Computer Architecture

Instructor

Dr. Dinh-Duc Anh-Vu

http://www.cse.hcmut.edu.vn/~anhvu

Chapter 6

VLIW PROCESSORS

Lecture 6 – VLIW Processors

Introduction and motivation

Loop unrolling

The IA-64 Architecture

Features of Itanium

What about Superscalar ?

It is a good architecture:

- The hardware solves everything
  - It detects potential parallelism between instructions;
  - It tries to issue as many instructions as possible in parallel;
  - It solves register renaming.
- Binary compatibility
  - If functional units are added in a new version of the architecture or some other improvements have been made to the architecture (without changing the instruction sets), old programs can benefit from the additional potential of parallelism.

    Because the new hardware will issue the old instruction sequence in a more efficient way.

What about Superscalar ?

But it has several problems:

- The architecture is very complex
  - Much hardware is needed for run-time detection.
  - There is a limit in how far we can go with this technique.
- The instruction window for execution is limited in size
  - This limits the capacity to detect potentially parallel instructions.

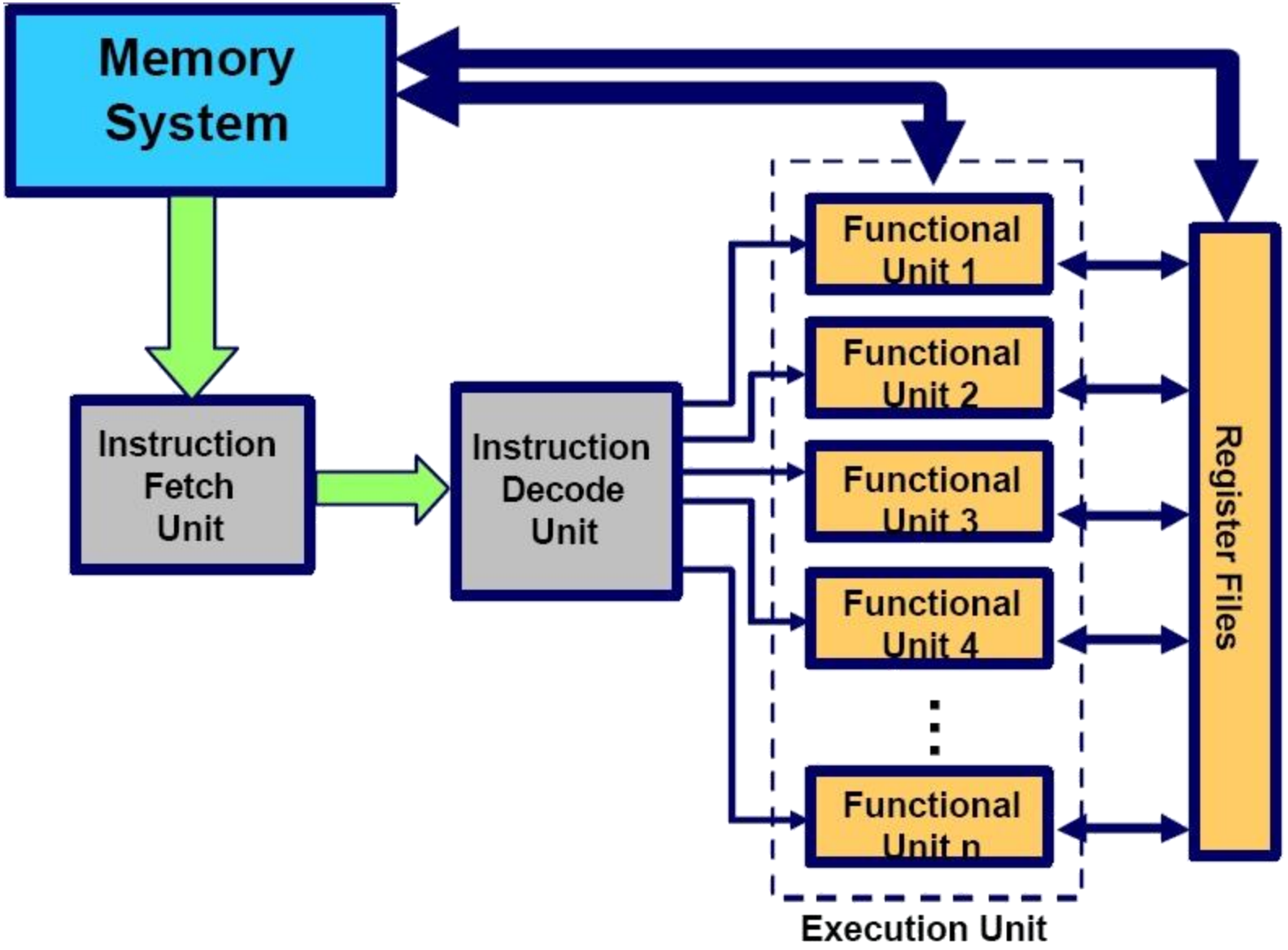
Very Long Instruction Word Processors

- In a VLIW (also called Very Large Instruction Word) processor, several operations that can be executed in parallel are placed an a single instruction word.

Instruction 1	Op1	Op2	Op3	Op4
Instruction 2	Op1	Ø	Op3	Op4
Instruction 3	Ø	Op2	Op3	Ø

- VLIW architectures rely on compile-time detection of parallelism.
  - The compiler analysis the program and detects operations to be executed in parallel.
- After one instruction has been fetched all the corresponding operations are issued in parallel.
  - No hardware is needed for run-time detection of parallelism.
- The instruction window problem is solved: the compiler can potentially analyze the whole program in order to detect parallel operations.

General Organization



Explicit Parallelism

- Instruction parallelism scheduled at compile time.
  - Included within the machine instructions
- Processor uses this info to perform parallel execution.
- Requires less complex hardware.
  - The number of FUs can be increased without needing additional sophisticated hardware to detect parallelism, as in superscalars.
- Compiler has much more time to determine possible parallel operations.
- Good compilers can detect parallelism based on global analysis of the whole program.

=> Explicit Parallel Instruction Computing (EPIC)

Main Problems

- A large number of registers is needed in order to keep all FUs active (to store operands and results).
- Large data transport capacity is needed between FUs and the register files and between register files and memory.
- High bandwidth between instruction cache and fetch unit.
  - Ex: one instruction with 8 operations, each 24 bits → 196 bits/instruction.
- Large code size, partially because unused operations → wasted bits in instruction word.
- Incompatibility of binary code
  - If a new version of the processor introduces additional FUs, the number of operations to execute in parallel is increased.
  - Therefore, the instruction word changes, and old binary code cannot be run on the new processor.

Lecture 6 – VLIW Processors

Introduction and motivation

Loop unrolling

The IA-64 Architecture

Features of Itanium

An Example

```
for (i=959; i>=0; i--)
    x[i] = x[i] + s;
```

Assumptions:

x is an array of floating point values;

s is a floating point constant.

Assumptions :

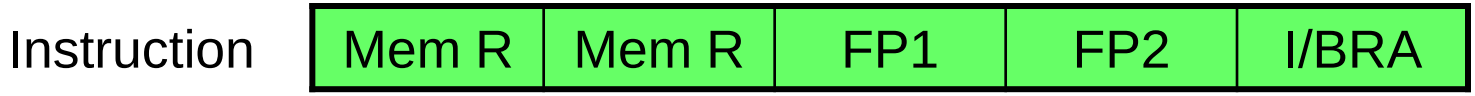
- R1 initially contains the address of the last element in x; the other elements are at lower addresses; x[0] is at address 0.
- Floating point register F2 contains the value s.
- A floating point value is 8 bytes long.

For an ordinary processor, this C code will be compiled to:

```
Loop: LDD   F0,(R1)      F0:=x[i];(load double)
      ADF   F4,F0,F2     F4:=F0+F2;(add floating point)
      STD   (R1),F4      x[i]:=F4;(store double)
      SBI   R1,R1,#8     R1:=R1-8;
      BGEZ  R1,Loop      branch if R1 = 0.
```

A VLIW Processor Example

- Two memory references, two FP operations, and one integer operation or branch can be packed in an instruction.



- The delay for a double word load is one additional clock cycle.
- The delay for a floating point operation is two additional clock cycles.
- No additional clock cycles for integer operations.

An Example (Cont'd)

```
Loop: LDD   F0,(R1)      load double
      ADF   F4,F0,F2     add FP
      STD   (R1),F4      store double
      SBI   R1,R1,#8
      BGEZ  R1,Loop
```

Note the displacement of 8 for R1 is needed, because we have already subtracted 8 from R1

Cycle				
1	LDD F0,(R1)			
2				
3		ADF F4,F0,F2		
4				
5				SBI R1,R1,#8
6	STD (R1+8),F4			BGEZ R1,Loop

- One iteration takes 6 cycles; the whole loop takes 960*6 = 5760 cycles.
- Almost no parallelism there; most of the fields in the instructions are empty.
- There are two completely empty cycles.

Loop Unrolling

Let us rewrite the previous example:

```
for (i=959; i>=0; i-=2){
   x[i] = x[i] + s;
   x[i-1] = x[i-1] + s;
}
```

- **Loop unrolling** is a technique used in compilers in order to increase the potential of parallelism in a program. This allows for more efficient code generation for processors with instruction level parallelism.

For an ordinary processor, this C code will be compiled to:

```
Loop: LDD   F0,(R1)      F0:=x[i];(load double)
      ADF   F4,F0,F2     F4:=F0+F2;(add floating pnt)
      STD   (R1),F4      x[i]:=F4;(store double)
      LDD   F6,(R1-8)    F6:=x[i-1];(load double)
      ADF   F8,F6,F2     F8:=F6+F2;(add floating pnt)
      STD   (R1-8),F8    x[i-1]:=F8;(store double)
      SBI   R1,R1,#16    R1:=R1-16;
      BGEZ  R1,Loop      branch if R1 = 0.
```

Loop Unrolling (2 iterations)

```
Loop: LDD   F0,(R1)      F0:=x[i];(load double)
      ADF   F4,F0,F2     F4:=F0+F2;(add floating pnt)
      STD   (R1),F4      x[i]:=F4;(store double)
      LDD   F6,(R1-8)    F6:=x[i-1];(load double)
      ADF   F8,F6,F2     F8:=F6+F2;(add floating pnt)
      STD   (R1-8),F8    x[i-1]:=F8;(store double)
      SBI   R1,R1,#16    R1:=R1-16;
      BGEZ  R1,Loop      branch if R1 = 0.
```

Cycle					
1	LDD F0,(R1)	LDD F6,(R1-8)			
2					
3			ADF F4,F0,F2	ADF F8,F6,F2	
4					
5					SBI R1,R1,#16
6	STD(R1+16),F4	STD(R1+8),F8			BGEZ R1,Loop

- There is an increased degree of parallelism in this case.
- We still have two completely empty cycles and many empty operation.
- However, we have a dramatic improvement in speed:
  - Two iterations take 6 cycles
  - The whole loop takes 480*6 = 2880 cycles

Loop Unrolling (3 iterations)

```
Loop: LDD   F0,(R1)      F0:=x[i];(load double)
      ADF   F4,F0,F2     F4:=F0+F2;(add floating pnt)
      STD   (R1),F4      x[i]:=F4;(store double)
      LDD   F6,(R1-8)    F6:=x[i-1];(load double)
      ADF   F8,F6,F2     F8:=F6+F2;(add floating pnt)
      STD   (R1-8),F8    x[i-1]:=F8;(store double)
      SBI   R1,R1,#16    R1:=R1-16;
      BGEZ  R1,Loop      branch if R1 = 0.
```

Cycle					
1	LDD F0,(R1)	LDD F6,(R1-8)			
2	LDD F10,(R1-16)				
3			ADF F4,F0,F2	ADF F8,F6,F2	
4			ADF F12,F10,F2		
5					
6	STD(R1),F4	STD(R1+8),F8			SBI R1,R1,#24
7	STD(R1+8),F12				BGEZ R1,Loop

- The degree of parallelism is further improved.
- There is still an empty cycle and many empty operations.
- Three iterations take 7 cycles
- The whole loop takes now 320*7 = 2240 cycles
  With 8 unrolled iteration, we have 120*9 = 1080 cycles

Loop Unrolling in General

- Given a certain set of resources (processor architecture) and a given loop, there is a limit on how many iterations should be unrolled.
  - Beyond that limit there is no gain any more.
- Loop unrolling increases the memory space needed to store the program.
- A good compiler has to find the optimal level of unrolling for each loop.
- The example before illustrates some of the *hardware support* needed to keep a VLIW processor busy:
  - Large number of registers (in order to store data for operations which are active in parallel);
  - Large traffic has to be supported in parallel:
    - register files ↔ memory
    - register files ↔ functional units

Lecture 6 – VLIW Processors

Introduction and motivation

Loop unrolling

The IA-64 Architecture

Features of Itanium

Some VLIW Processors

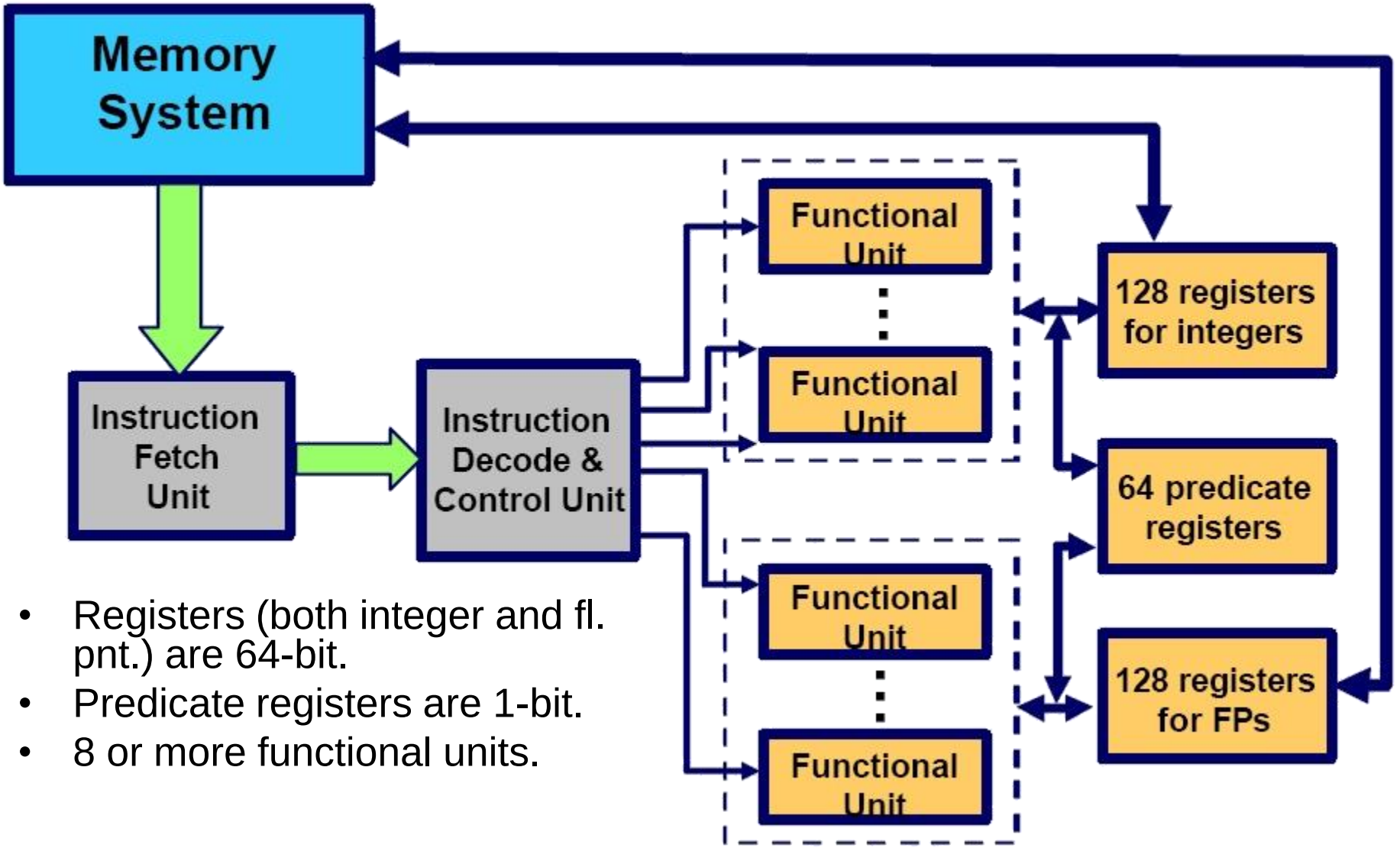
- Successful current VLIW processors:
  - TriMedia of Philips
  - TMS320C6x of Texas Instruments

  Both are targeting the multimedia market.
- The IA-64 architecture from Intel and Hewlett-Packard.
  - This family uses many of the VLIW ideas.
  - It is not "just" a multimedia processor, but intended to become "the new generation" processor for servers and workstations.
  - The first product in 2001: Itanium

The IA-64 Architecture

- IA-64 is not a pure VLIW architecture, but many of its features are typical for VLIW processors.
- These are typical VLIW features of IA-64:
  - Instruction-level parallelism fixed at compile-time.
  - (Very) long instruction word.
- These are specific for the IA-64 Architecture:
  - Branch predication (NOT prediction!).
  - Speculative loading
- Intel calls the Itanium an EPIC (explicitly parallel instruction computing) processor, because the parallelism of operations is explicitly there in the instruction word.

IA-64 Basic architecture

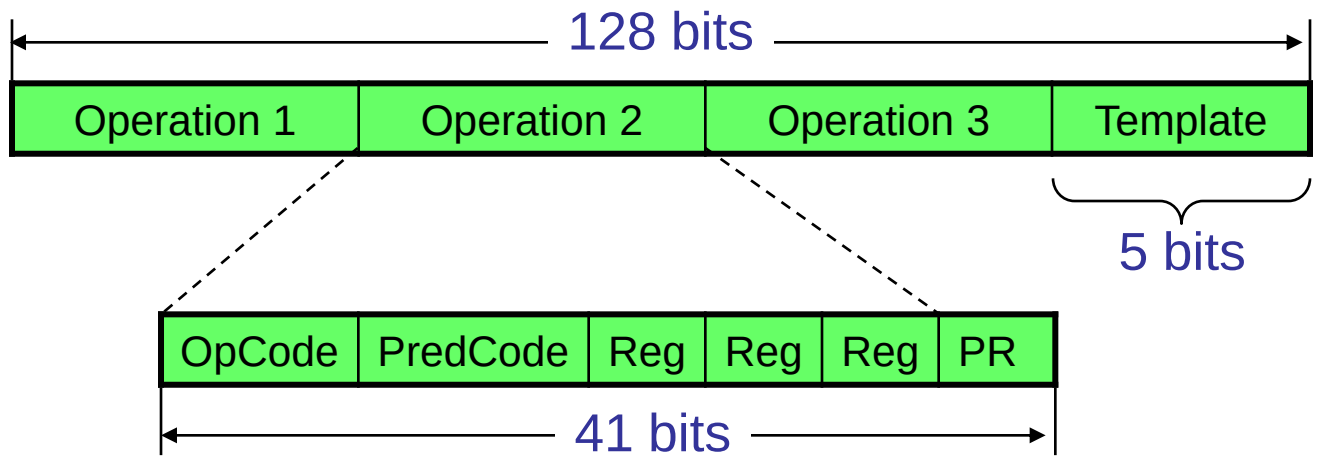


- Registers (both integer and fl. pnt.) are 64-bit.
- Predicate registers are 1-bit.
- 8 or more functional units.

IA-64 Functional Units

- I-Unit
  - Integer arithmetic
  - Shift and add
  - Logical
  - Compare
  - Integer multimedia ops
- M-Unit
  - Load and store
    - Between register and memory
  - Some integer ALU
- B-Unit
  - Branch instructions
- F-Unit
  - Floating point instructions

Instruction Format



- 128 bit bundle
  - Holds three instructions (syllables) plus template
    - The three operations in the instruction are not necessarily to be executed in parallel!
  - Can fetch one or more bundles at a time
  - Template contains info on which instructions can be executed in parallel
  - Instruction is 41 bit long
    - More registers than usual RISC
    - Predicated execution registers

Instruction Format (cont'd)

- The template provides high flexibility and avoids several problems with classical VLIW processors:
  - Operations in one instruction don't have to be parallel.
    - Can mix dependent and independent instructions in same bundle
    - no places have to be left empty when no operation is there to be executed in parallel.
  - The number of operations to be executed in parallel is not restricted by the instruction size (due to the template)
    - e.g. a stream of 8 instructions may be executed in parallel
    - Compiler will have re-ordered instructions to form contiguous bundles
    - successive processor generations can have different number of functional units without changing the instruction word.
    - improved binary compatibility.
  - If, according to the template, more operations can be executed in parallel than functional units available, the processor is able to take them sequentially.

Template Field Encoding and Instruction Set Mapping

- The field specifies the mapping of instruction slots to functional unit types. Not all possible mappings of instructions to units are available
- The field indicates the presence of any stops. A stop indicates to the hardware that one or more instructions before the stop may have certain kinds of resource dependencies with one or more instructions after the stop.

Template	Slot 0	Slot 1	Slot 2
00	M-unit	I-unit	I-unit
01	M-unit	I-unit	I-unit
02	M-unit	I-unit	I-unit
03	M-unit	I-unit	I-unit
04	M-unit	L-unit	X-unit
05	M-unit	L-unit	X-unit
08	M-unit	M-unit	I-unit
09	M-unit	M-unit	I-unit
0A	M-unit	M-unit	I-unit
0B	M-unit	M-unit	I-unit
0C	M-unit	F-unit	I-unit
0D	M-unit	F-unit	I-unit
0E	M-unit	M-unit	F-unit
0F	M-unit	M-unit	F-unit
10	M-unit	I-unit	B-unit
11	M-unit	I-unit	B-unit
12	M-unit	B-unit	B-unit
13	M-unit	B-unit	B-unit
16	B-unit	B-unit	B-unit
17	B-unit	B-unit	B-unit
18	M-unit	M-unit	B-unit
19	M-unit	M-unit	B-unit
1C	M-unit	F-unit	B-unit
1D	M-unit	F-unit	B-unit

Assembly Language Format

- `[qp] mnemonic [.comp] dest = srcs //`
- `qp` - predicate register
  - 1 at execution then execute and commit result to hardware
  - 0 result is discarded
- `mnemonic` - name of instruction
- `comp` – one or more instruction completers used to qualify mnemonic
- `dest` – one or more destination operands
- `srcs` – one or more source operands
- `//` - comment
- Instruction groups and stops indicated by `;;`
  - Sequence without read after write or write after write
  - Do not need hardware register dependency checks

Assembly Examples

```
ld8 r1 = [r5] ;; //first group
add r3 = r1, r4 //second group
```

- Second instruction depends on value in r1
  - Changed by first instruction
  - Can not be in same group for parallel execution

```
ld8 r1 = [r5]          // first group
sub r6 = r8, r9 ;; // first group
add r3 = r1, r4        // second group
st8 [r6] = r12         // second group
```

Instruction-level Parallelism in IA-64

- Predication execution
- Control speculation
- Data speculation
- Software pipelining

Predicated Execution

- Any operation can refer to a predicate register.

  `<Pi> operation` where *i* is the number of a predicate register (between 0 and 63)

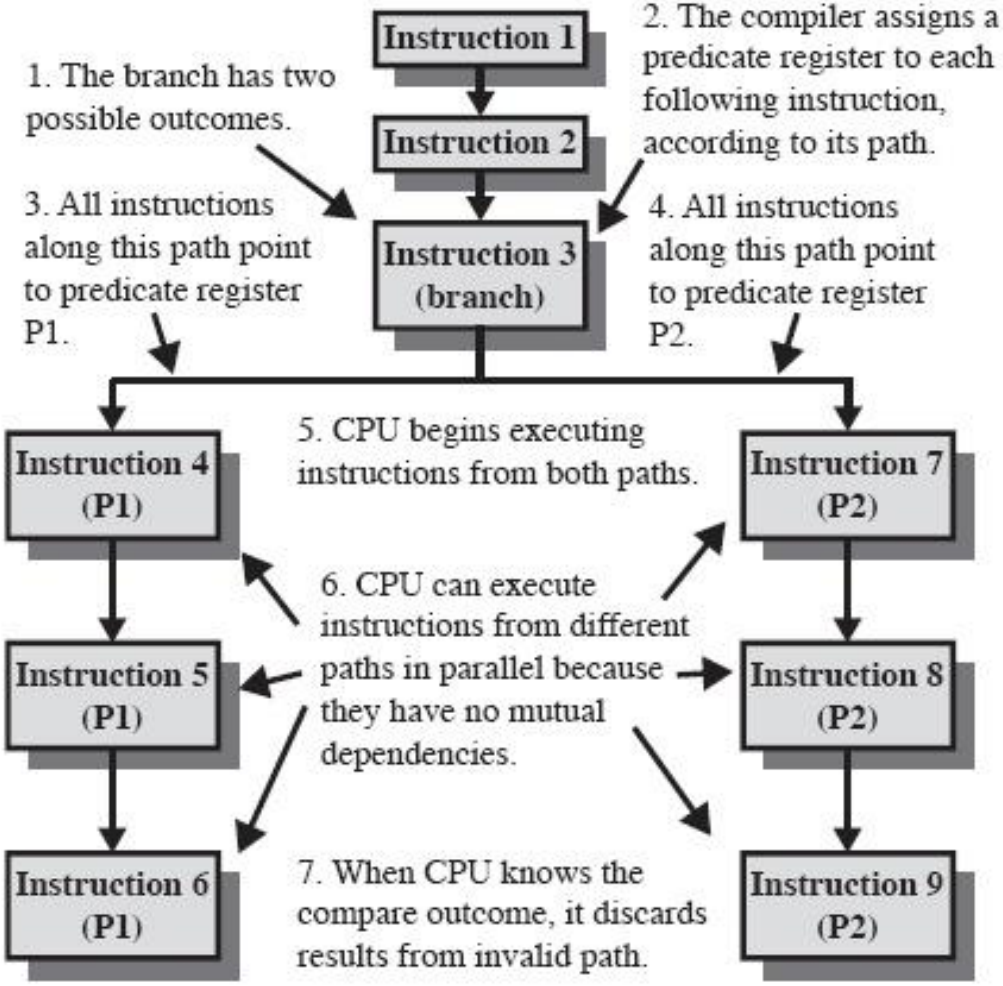
- This means that an operation is to be committed (the results made visible) only when the respective predicate is true (the predicate register gets value 1).
- If the predicate value is known when the operation is issued, the operation is executed only if this value is true.
- If the predicate is not known at that moment, the operation is started; if the predicate turns out to be false, the operation is discarded.
- If no predicate register is mentioned, the operation is executed and committed unconditionally.

Branch Predication

- **Branch predication** is an aggressive compilation technique for generation of code with high degree of instruction level parallelism.
- In order to exploit all potential of parallelism in the machine it lets operations from both branches of a conditional branch to be executed in parallel.
- In this way, branches are eliminated and replaced by conditional execution.
- In order to do this hardware support is needed.
- Branch predication is based on the instructions for predicated execution provided by the IA-64 architecture.

- **The idea is**: let instructions from both branches go on in parallel, before the branch condition has been evaluated. The hardware (predicated execution) takes care that only those corresponding to the right branch will be committed.

Branch Predication



The compiler might rearrange instructions in this order, pairing instructions 4 and 7, 5 and 8, and 6 and 9 for parallel execution.

Instruction 1	Instruction 2	Instruction 3
Instruction 4	Instruction 7	Instruction 5
Instruction 8	Instruction 6	Instruction 9

Branch Predication (cont'd)

Branch predication is not branch prediction!

- Branch prediction:

  Guess for one branch and then go along that one; if the guess is bad, undo all the work, and go to the right one.

- Branch predication:

  Both branches are started and when the condition is known (the predicate registers are set) the right instructions are committed, the others are discarded.

  There is no lost time with failed predictions.

  (You pay by introducing duplicated hardware.)

Branch Predication Example

```
if (a&&b)
      j = j + 1;
else
      if (c)
            k = k + 1;
      else
            k = k – 1;
i = i + 1;
```

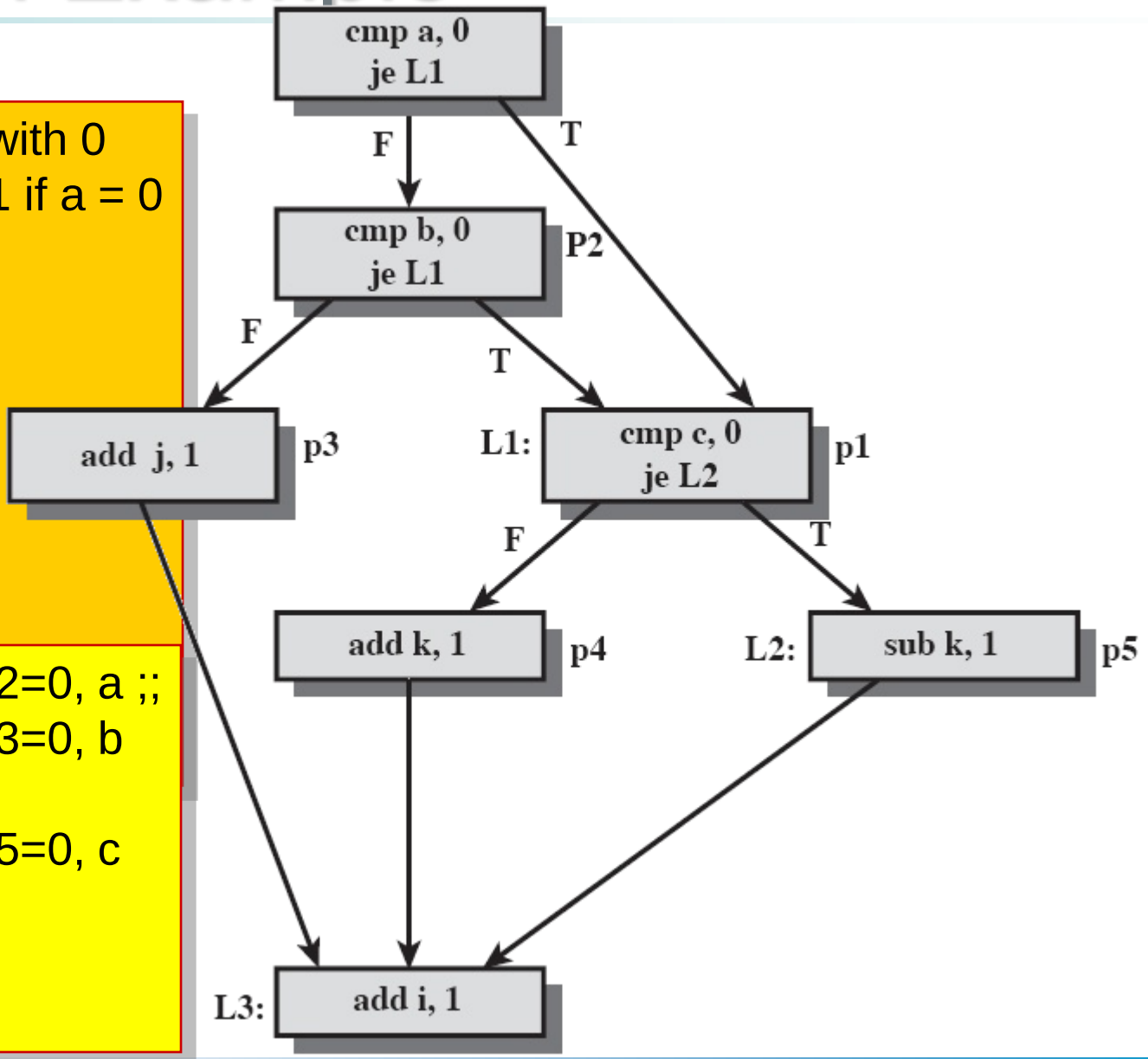
Pentium assembly

```
      cmp a, 0       ; compare a with 0
      je L1          ; branch to L1 if a = 0
      cmp b, 0
      je L1
      add j, 1       ; j = j + 1
      jmp L3
L1:   cmp c, 0
      je L2
      add k, 1       ; k = k + 1
      jmp L3
L2:   sub k, 1       ; k = k – 1
L3:   add i, 1       ; i = i + 1
```

Predication Example

```
      cmp a, 0      ; compare a with 0
      je L1         ; branch to L1 if a = 0
      cmp b, 0
      je L1
      add j, 1      ; j = j + 1
      jmp L3
L1:   cmp c, 0
      je L2
      add k, 1      ; k = k + 1
      jmp L3
L2:   sub k,
L3:   add i,
```

```
      cmp. eq p1, p2=0, a ;;
(p2)  cmp. eq p1, p3=0, b
(p3)  add j=1, j
(p1)  cmp. ne p4, p5=0, c
(p4)  add k=1, k
(p5)  add k= – 1, k
      add i=1, i
```



Control & Data Speculation

- Control
  - AKA Speculative loading
  - Load data from memory before needed
- Data
  - Load moved before store that might alter memory location
  - Subsequent check in value

Placement of Loading

- A load from memory operation can be placed so that memory latency is avoided (the value is already there when it's needed):

```
load r1, X   // load from address X into r1
add r2, r1   // r2 = r2 + r1
add r4, r3   // r4 = r4 + 43
sub r2, r4   // r2 = r2 - r4
```

```
load r1, X   // load from address X into r1
add r4, r3   // r4 = r4 + 43
add r2, r1   // r2 = r2 + r1
sub r2, r4   // r2 = r2 - r4
```

Speculative Loading

- What if a load is moved across a branch?
  - Ex. from within a if-branch to before the conditional branch.
- The load will be executed for both branches.
  - This shouldn't be a problem if hardware resources are available.
  - However, if, for example, a page fault is generated. Handling such an exception takes extremely much time and resources.
    - A whole page has to be brought in from the secondary memory to the main memory.
  - If the load will turn out to be not needed, it is wasting time.
- We would like to handle the exception only when we really need the loaded value!
- This is solved by speculative loading!

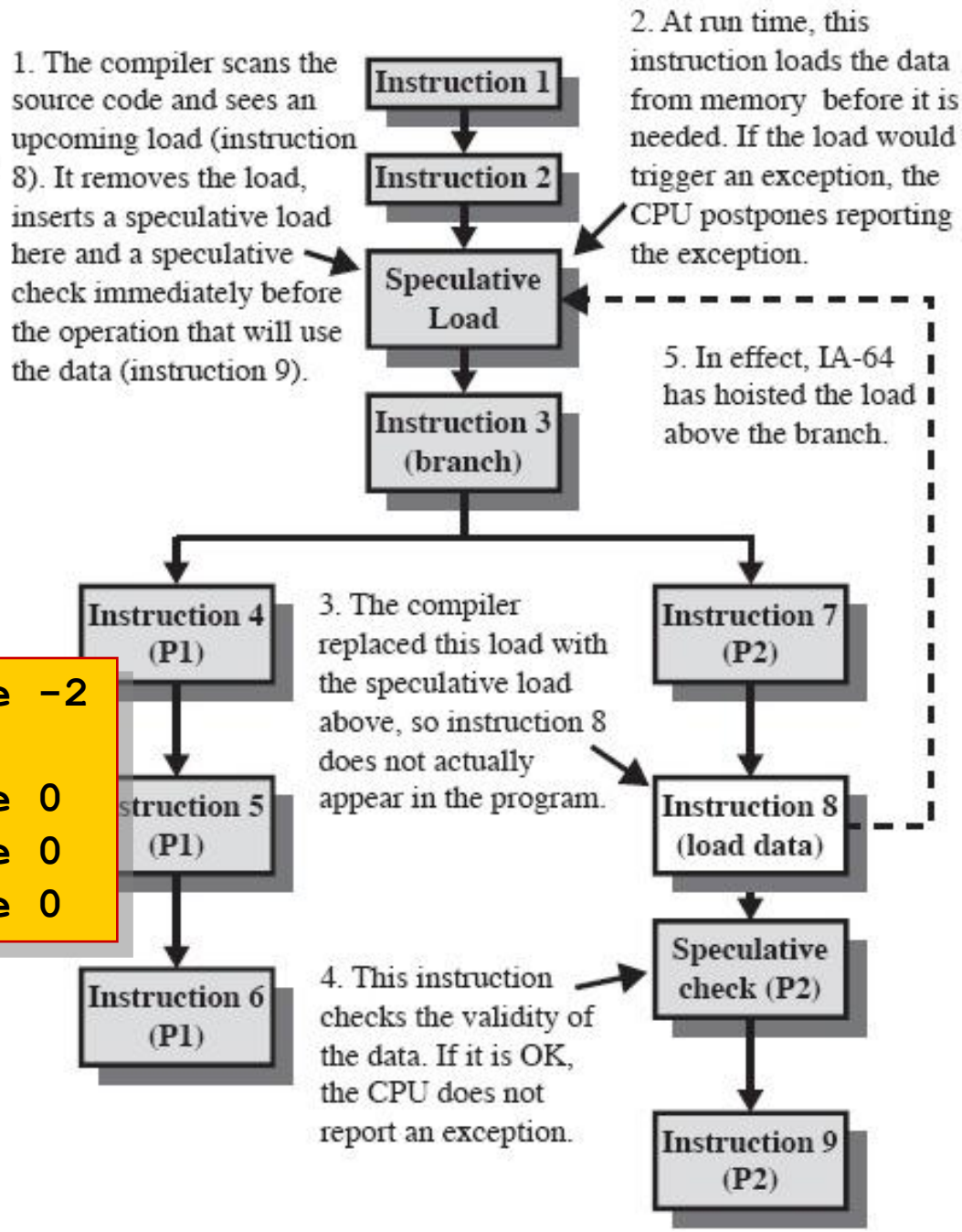
Speculative Loading (cont'd)

- With speculative loading a load instruction in the original program can be replaced by two instructions:
  - A speculative load (**`ld.s`**) which performs the memory fetch and detects an exception (like page fault) if generated. The exception, however is not signaled to the operating system.
    - The speculative load is the one which is moved if needed for latency reduction.
  - A checking instruction (**`chk.s`**) which signals the exception if one has been detected by the speculative load.
    - If no exception has been detected, nothing happens.
    - The checking instruction remains in place (is not moved).
- By this technique, even if loads are moved across branch boundaries, exceptions are handled only on the branch which really needs the value.

Speculative Loading Example

```
(p1) br some_label    // cycle 0
     ld8 r1 = [r5] ;; // cycle 1
     add r2 = r1, r3  // cycle 3
```

```
     ld8.s r1 = [r5] ;;    // cycle -2
     // other instructions
(p1) br some_label         // cycle 0
     chk.s r1, recovery    // cycle 0
     add r2 = r1, r3       // cycle 0
```



Data Speculation

- Load moved before store that might alter memory location
- Subsequent check in value

```
st8 [r4] = r12      // cycle 0
ld8 r6 = [r8] ;;    // cycle 0
add r5 = r6, r7 ;; // cycle 2
st8 [r18] = r5      // cycle 3
```

```
ld8.a r6, [r8] ;;  // cycle -2 or earlier;
                   // advance load
// other instructions
st8 [r4] = r12     // cycle 0
ld8.c r6 = [r8]    // cycle 0; check load
add r5 = r6, r7 ;; // cycle 0
st8 [r18] = r5     // cycle 1
```

Data Speculation (cont'd)

```
    ld8.a r6, [r8] ;;  // cycle -3 or earlier;
                       // advance load
    // other instructions
    add r5 = r6, r7 ;; // cycle -1; add that uses r6
    // other instructions
    st8 [r4] = r12     // cycle 0
    chk.a r6, recover  // cycle 0; check
back:                  // return point from jump to recover
    st8 [r18] = r5     // cycle 1
```

```
recover:
    ld8 r6, [r8] ;;    // reload r6 from [r8]
    add r5 = r6, r7 ;; // re-execute the add
    br back            // jump back to main code
```

Software Pipelining

```
L1:    ld4 r4=[r5],4 ;;   //cycle 0 load postinc 4
       add r7=r4,r9 ;;    //cycle 2
       st4 [r6]=r7,4      //cycle 3 store postinc 4
       br.cloop L1 ;;     //cycle 3
```

- Adds constant to one vector and stores result in another
- No opportunity for instruction level parallelism
- Instruction in iteration *x* all executed before iteration *x+1* begins
- If no address conflicts between loads and stores can move independent instructions from loop *x+1* to loop *x*

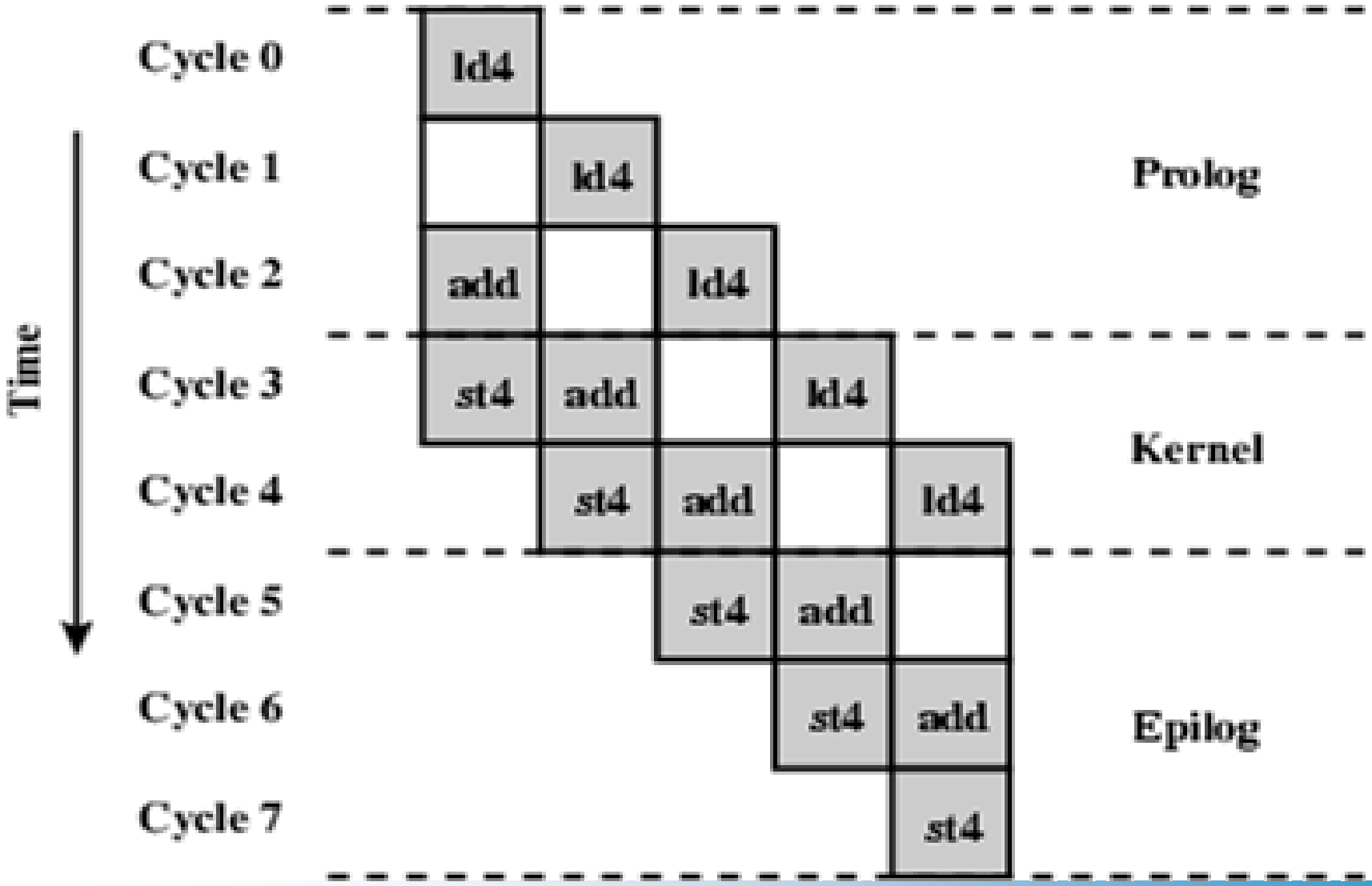
Unrolled Loop

```
ld4 r32=[r5],4;; //cycle 0
ld4 r33=[r5],4;; //cycle 1
ld4 r34=[r5],4   //cycle 2
add r36=r32,r9;; //cycle 2
ld4 r35=[r5],4   //cycle 3
add r37=r33,r9   //cycle 3
st4 [r6]=r36,4;; //cycle 3
ld4 r36=[r5],4   //cycle 3
add r38=r34,r9   //cycle 4
st4 [r6]=r37,4;; //cycle 4
add r39=r35,r9   //cycle 5
st4 [r6]=r38,4;; //cycle 5
add r40=r36,r9   //cycle 6
st4 [r6]=r39,4;; //cycle 6
st4 [r6]=r40,4;; //cycle 7
```

Unrolled Loop Detail

- Completes 5 iterations in 7 cycles
  - Compared with 20 cycles in original code
- Assumes two memory ports
  - Load and store can be done in parallel

Software Pipeline Example Diagram



Support For Software Pipelining

- Automatic register renaming
  - Fixed size are of predicate and fp register file (p16-P32, fr32-fr127) and programmable size area of gp register file (max r32-r127) capable of rotation
  - Loop using r32 on first iteration automatically uses r33 on second
- Predication
  - Each instruction in loop predicated on rotating predicate register
    - Determines whether pipeline is in prolog, kernel or epilog
- Special loop termination instructions
  - Branch instructions that cause registers to rotate and loop counter to decrement

Superscalar vs. IA-64

Superscalar	IA-64
RISC-like instructions, one per word	RISC-like instructions bundled into groups of three
Multiple parallel execution units	Multiple parallel execution units
Reorders and optimizes instruction stream at run time	Reorders and optimizes instruction stream at compile time
Branch prediction with speculative execution of one path	Speculative execution along both paths of a branch
Loads data from memory only when needed, and tries to find the data in the caches first	Speculatively loads data before its needed, and still tries to find data in the caches first

Lecture 6 – VLIW Processors

Introduction and motivation

Loop unrolling

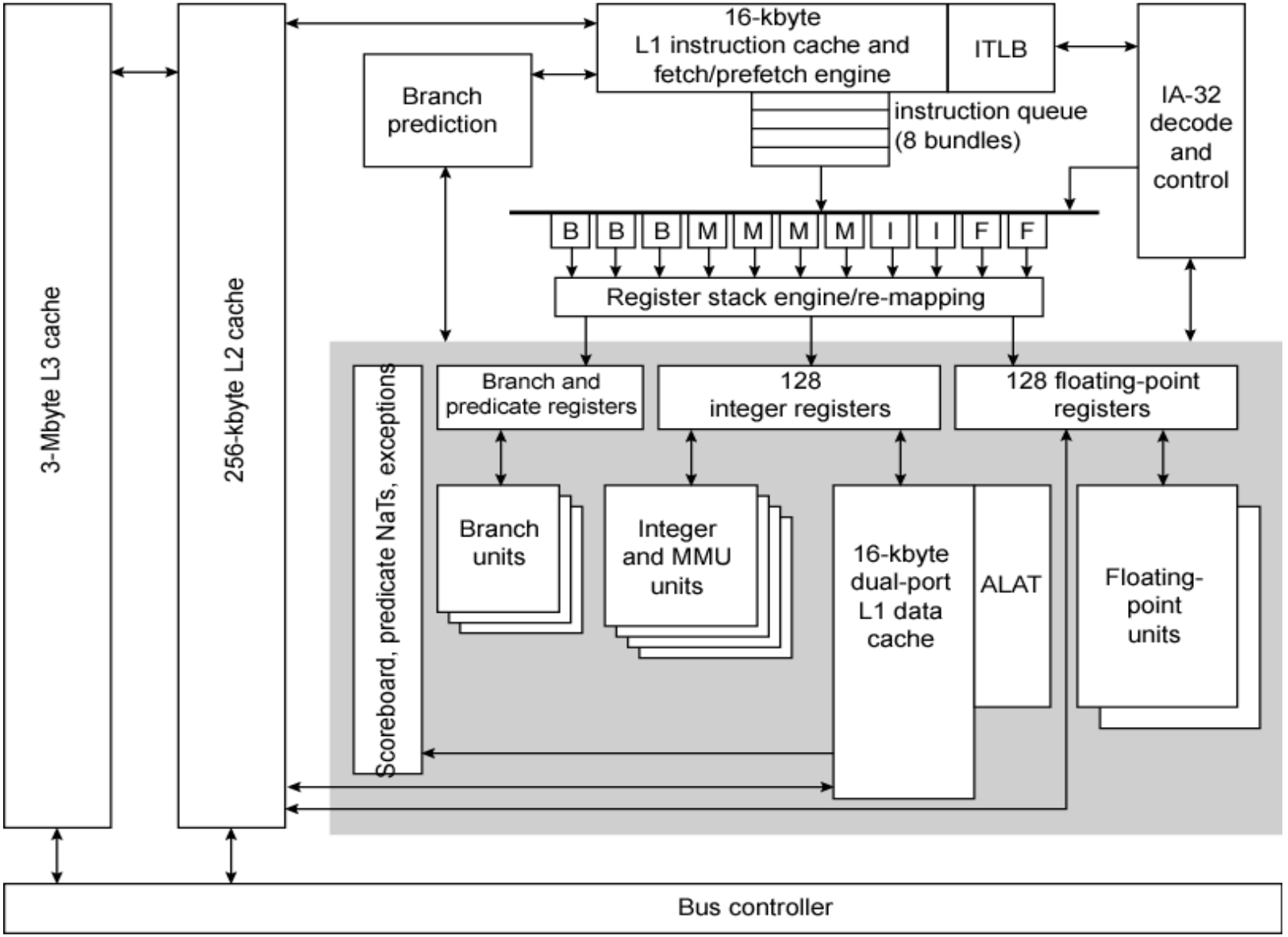
The IA-64 Architecture

Features of Itanium

Itanium Organization

- Superscalar features
  - Six wide, ten stage deep hardware pipeline
  - Dynamic prefetch
  - Branch prediction
  - Register scoreboard to optimise for compile time nondeterminism
- EPIC features
  - Hardware support for predicated execution
  - Control and data speculation
  - Software pipelining

Itanium 2 Processor Diagram



Itanium 2 (1)

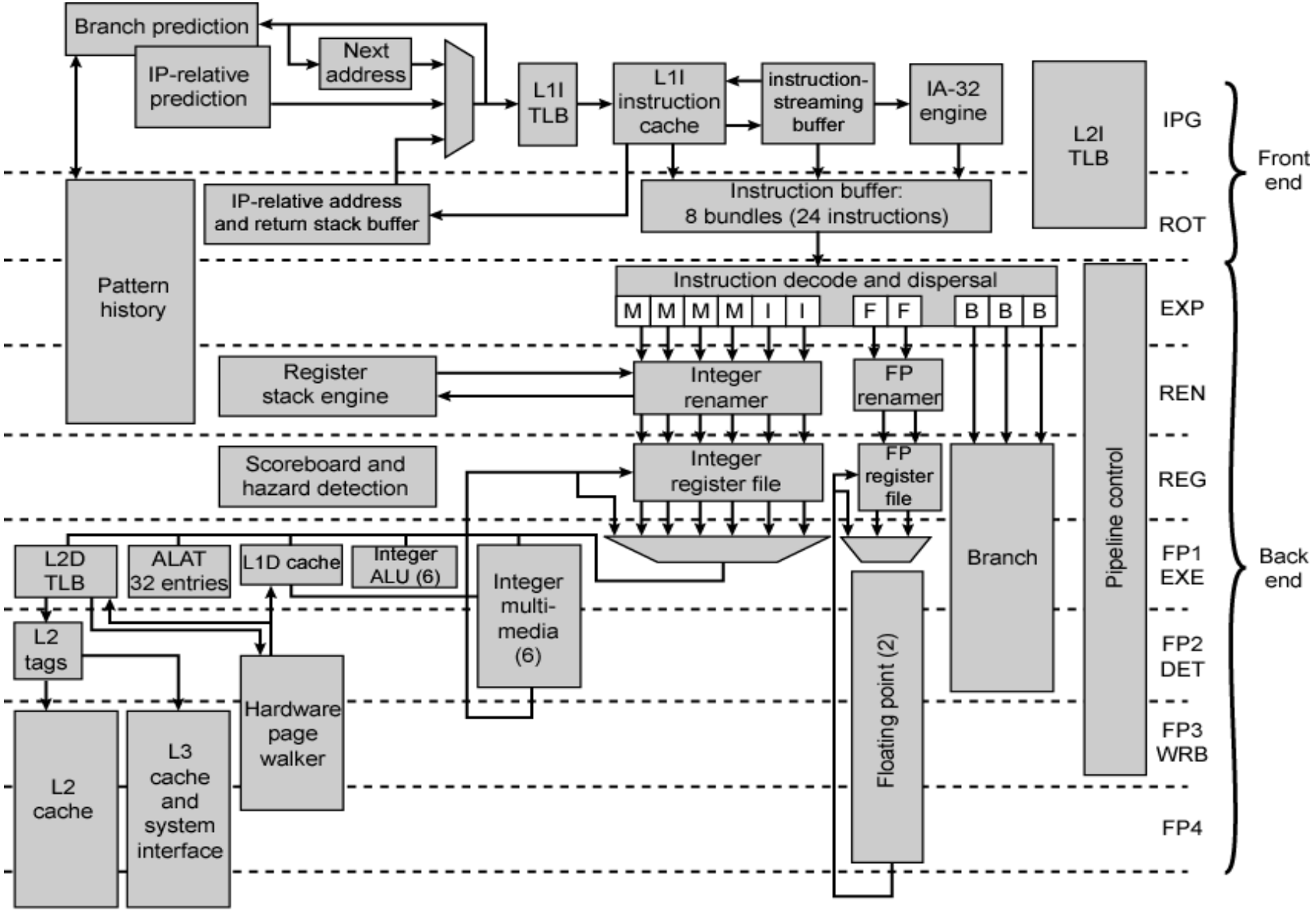
- 8-stage pipeline
  - All but floating-point instructions
- Pipeline stages
  - Instruction pointer generation (IPG)
    - Delivers and instruction pointer to L1I cache
  - Instruction rotation (ROT)
    - Fetch instructions and rotate into position
    - Bundle 0 contains first instruction to be executed
  - Instruction template decode, expand and disperse (EXP)
    - Decode instruction templates
    - Disperse up to 6 instructions through 11 ports in conjunction with opcode information for execution units
  - Rename and decode (REN)
    - Rename (remap) registers for register stack engine
    - Decode instructions
  - Register file read (REG)
    - Delivers operands to execution units

Itanium 2 (2)

- ALU execution (EXE)
  - Execute operations
- Last stage for exception detection (DET)
  - Detect exceptions
  - Abandon result if instruction predicate not true
  - Resteer mispredicted branches
- Write back (WRB)
  - Write results back to register file

- Floating-point instructions
  - First five stages same
  - Followed by four floating-point pipeline stages
  - Followed by write-back stage

Itanium 2 Processor Pipeline



Summary

- In order to keep up with increasing demands on performance, superscalar architectures become excessively complex.
- VLIW architectures avoid complexity by relying exclusively on the compiler for parallelism detection.
- Operation level parallelism is explicit in the instructions of a VLIW processor.
- Not having to deal with parallelism detection, VLIW processors can have a high number of functional units. This, however, generates the need for a large number of registers and high communication bandwidth.
- In order to keep the large number of functional units busy, compilers for VLIW processors have to be aggressive in parallelism detection.
- Loop unrolling is used by compilers in order to increase the degree of parallelism in loops: several iterations of a loop are unrolled and handled in parallel.

Summary (Cont'd)

- The IA-64 family uses VLIW techniques for the future generation of high-end processors. Itanium is the first member of this family.
- The instruction format avoids empty operations and allows more flexibility in parallelism specification, by using the template field.
- Beside typical VLIW features, the Itanium architecture uses branch predication and speculative loading.
- Branch predication is based on predicated execution. The execution of an instruction can be connected to a predicate register. The instruction will be committed only if the predicate becomes true.
  - Branch predication allows instructions from different branches to be started in parallel. Finally, only those in the right branch will be committed.
- Speculative loading tries to reduce latencies generated by load instructions. It allows load instructions to be moved across branch boundaries, without page exceptions being handled if not needed.