High Performance Architectures

EPIC

Part 2

EPIC: a paradigm shift

- Superscalar RISC solution
  - Based on sequential execution semantics
  - Compiler’s role is limited by the instruction set architecture
  - Superscalar hardware identifies and exploits parallelism

- EPIC solution – (the evolution of VLIW)
  - Based on parallel execution semantics
  - EPIC ISA enhancements support static parallelization
  - Compiler takes greater responsibility for exploiting parallelism
  - Compiler / hardware collaboration often resembles superscalar

EPIC: a paradigm shift

- Advantages of pursuing EPIC architectures
  - Make wide issue & deep latency less expensive in hardware
  - Allow processor parallelism to scale with additional VLSI density
- Architect the processor to do well with in-order execution
  - Enhance the ISA to allow static parallelization
  - Use compiler technology to parallelize program
  - However, a purely static VLIW is not appropriate for general-purpose use

The fusion of VLIW and superscalar techniques

- **Superscalars** need improved support for static parallelization
  - Static scheduling
  - Limited support for predicated execution
- **VLIWs** need improved support for dynamic parallelization
  - Caches introduce dynamically changing memory latency
  - Compatibility: issue width and latency may change with new hardware
  - Application requirements - e.g. object oriented programming with dynamic binding
- **EPIC processors** exhibit features derived from both
  - Interlock & out-of-order execution hardware are compatible with EPIC (but not required!)
  - EPIC processors can use dynamic translation to parallelize in software

Many EPIC features are taken from VLIWs

- Minisupercomputer products stimulated VLIW research (FPS, Multiflow, Cydrome)
  - Minisupercomputers were specialized, costly, and short-lived
  - Traditional VLIWs not suited to general purpose computing
  - VLIW resurgence in single chip DSP & media processors
- Minisupercomputers exaggerated forward-looking challenges:
  - Long latency
  - Wide issue
  - Large number of architected registers
  - Compile-time scheduling to exploit exotic amounts of parallelism
- EPIC exploits many VLIW techniques

Shortcomings of early VLIWs

- Expensive multi-chip implementations
- No data cache
- Poor "scalar" performance
- No strategy for object code compatibility

EPIC design challenges

- Develop architectures applicable to general-purpose computing
  - Find substantial parallelism in ”difficult to parallelize” scalar programs
  - Provide compatibility across hardware generations
  - Support emerging applications (e.g. multimedia)
- Compiler must find or create sufficient ILP
- Combine the best attributes of VLIW & superscalar RISC
  (incorporated best concepts from all available sources)
- Scale architectures for modern single-chip implementation

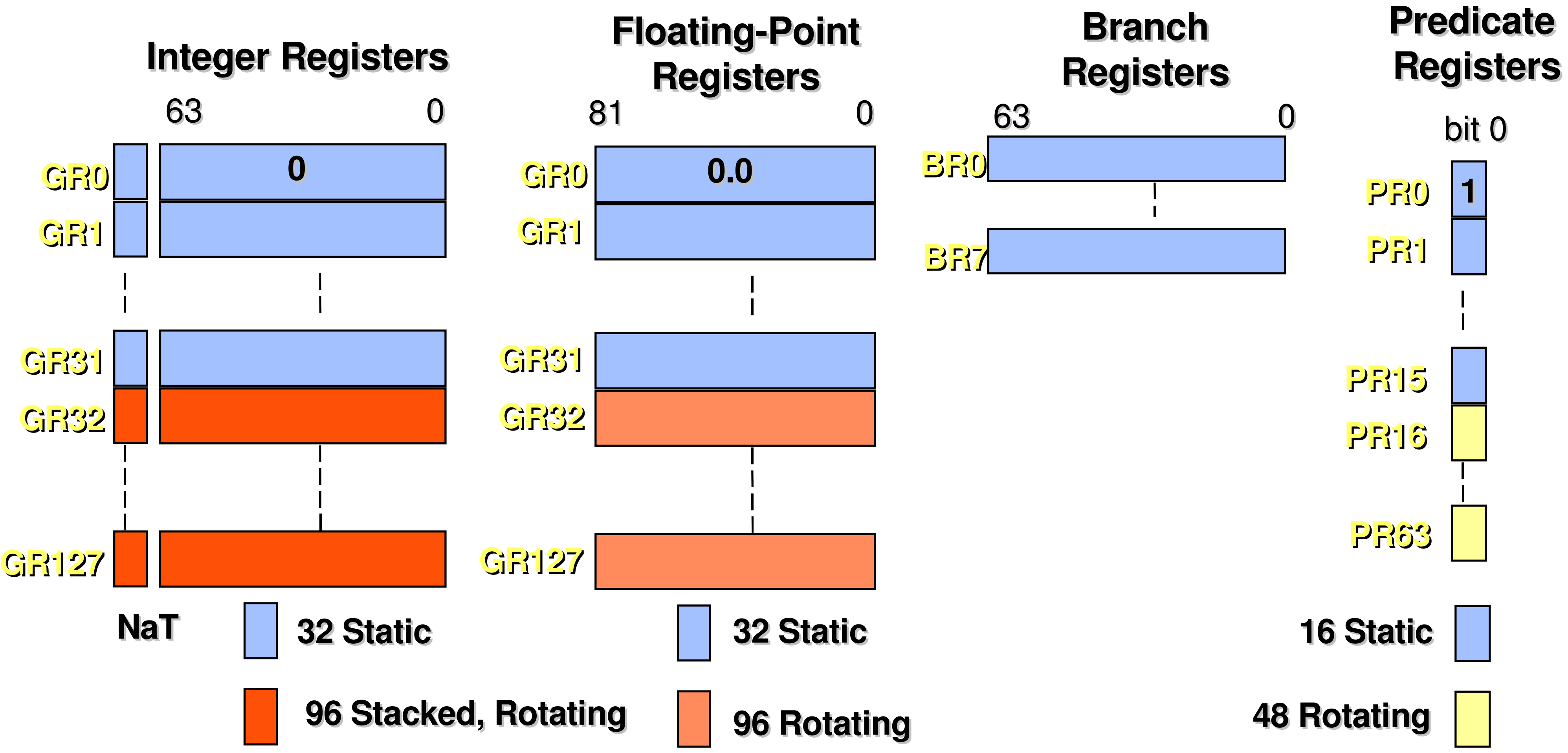
EPIC Processors, Intel's IA-64 ISA and Itanium

- Joint R&D project by Hewlett-Packard and Intel (announced in June 1994)
- This resulted in **explicitly parallel instruction computing (EPIC)** design style:
  - specifying ILP explicit in the machine code, that is, the parallelism is encoded directly into the instructions similarly to VLIW;
  - a fully predicated instruction set;
  - an inherently scalable instruction set (i.e., the ability to scale to a lot of FUs);
  - many registers;
  - speculative execution of load instructions

IA-64 Architecture

- Unique architecture features & enhancements
  - Explicit parallelism and templates
  - Predication, speculation, memory support, and others
  - Floating-point and multimedia architecture
- IA-64 resources available to applications
  - Large, application visible register set
  - Rotating registers, register stack, register stack engine
- IA-32 & PA-RISC compatibility models

IA-64's Large Register File



Intel's IA-64 ISA

- IA-64 **instructions** are 41-bit (previously stated 40 bit) long and consist of
  - op-code,
  - predicate field (6 bits),
  - two source register addresses (7 bits each),
  - destination register address (7 bits), and
  - special fields (includes integer and floating-point arithmetic).
- The 6-bit predicate field in each IA-64 instruction refers to a set of 64 predicate registers.
- 6 types of instructions
  - A: Integer ALU ==> I-unit or M-unit
  - I: Non-ALU integer ==> I-unit
  - M: Memory ==> M-unit
  - B: Branch ==> B-unit
  - F: Floating-point ==> F-unit
  - L: Long Immediate ==> I-unit
- IA-64 instructions are packed by compiler into **bundles**.

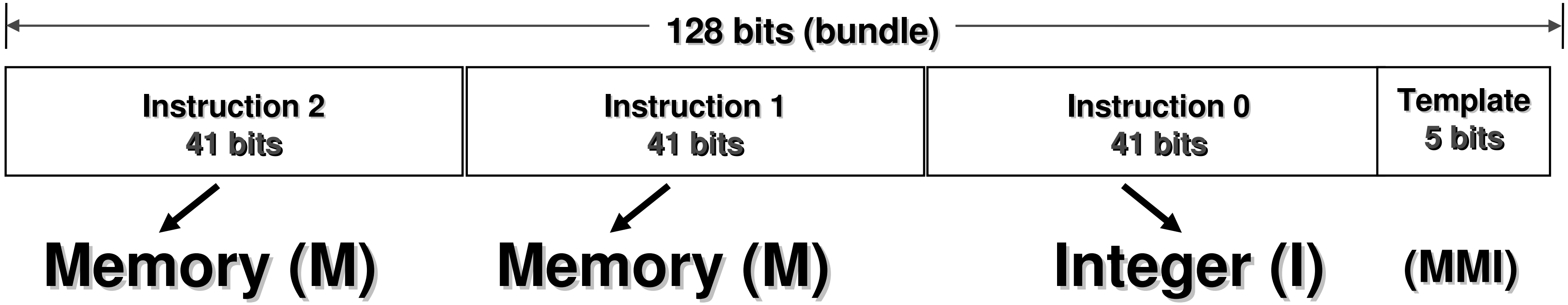
IA-64 Bundles

- A *bundle* is a 128-bit long instruction word (LIW) containing three 41-bit IA-64 instructions along with a so-called 5-bit *template* that contains instruction grouping information
- IA-64 does not insert no-op instructions to fill slots in the bundles

- The *template* explicitly indicates (ADAG):
  - first 4 bits: types of instructions
  - last bit (stop bit): whether the bundle can be executed in parallel with the next bundle
  - (previous literature): whether the instructions in the bundle can be executed in parallel or if one or more must be executed serially (no more in ADAG description)
- Bundled instructions don't have to be in their original program order, and they can even represent entirely different paths of a branch

- Also, the compiler can mix dependent and independent instructions together in a bundle, because the template keeps track of which is which

IA-64 : Explicitly Parallel Architecture



- IA-64 template specifies
  - The type of operation for each instruction
    - MFI, MMI, MII, MLI, MIB, MMF, MFB, MMB, MBB, BBB
  - Intra-bundle relationship
    - M / MI or MI / I
  - Inter-bundle relationship
- Most common combinations covered by templates
  - Headroom for additional templates
- Simplifies hardware requirements
- Scales compatibly to future generations

M=Memory
F=Floating-point
I=Integer
L=Long Immediate
B=Branch

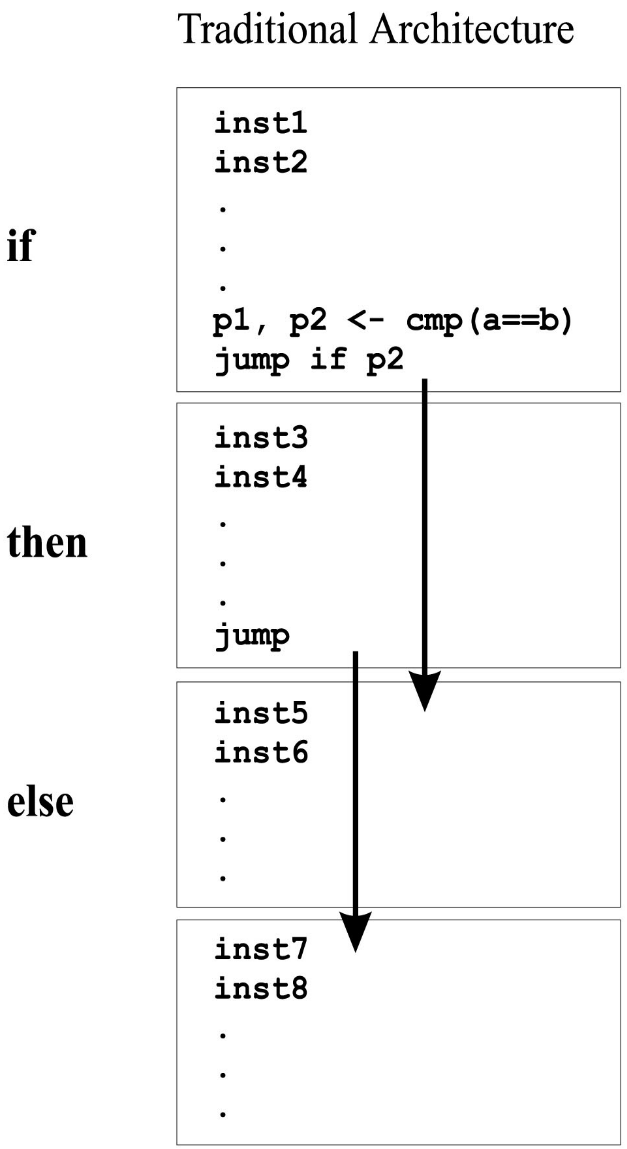
IA-64 Scalability

- A single bundle containing three instructions corresponds to a set of three FUs.
- If an IA-64 processor had *n* sets of three FUs each then using the template information it would be possible to chain the bundles to create instruction word of *n* bundles in length.
- This is the way to provide scalability of IA-64 to any number of FUs.

Predication in IA-64 ISA

- Branch prediction: paying a heavy penalty in lost cycles if mispredicted.
- IA-64 compilers uses predication to remove the penalties caused by mispredicted branches and by the need to fetch from noncontiguous target addresses by jumping over blocks of code beyond branches.
- When the compiler finds a branch statement it marks all the instructions that represent each path of the branch with a unique identifier called a **predicate**.
- IA-64 defines a 6-bit field (predicate register address) in each instruction to store this predicate. ==> 64 unique predicates available at one time.
- Instructions that share a particular branch path will share the same predicate.

If-then-else statement



Traditional Architecture

if

```
inst1
inst2
.
.
.
p1, p2 <- cmp(a==b)
jump if p2
```

then

```
inst3
inst4
.
.
.
jump
```

else

```
inst5
inst6
.
.
.
```

```
inst7
inst8
.
.
.
```

(a)

EPIC Architecture

if

```
      inst1
      inst2
      .
      .
      .
      p1, p2 <- cmp(a==b)
```

then

```
(p1)  inst3
(p1)  inst4
      .
      .
      .
```

else

```
(p2)  inst5
(p2)  inst6
      .
      .
      .

      inst7
      inst8
      .
      .
      .
```

(b)

Predication in IA-64 ISA

- At run time, the CPU scans the templates, picks out the independent instructions, and issues them in parallel to the FUs.

- Predicated branch: the processor executes the code for every possible branch outcome.

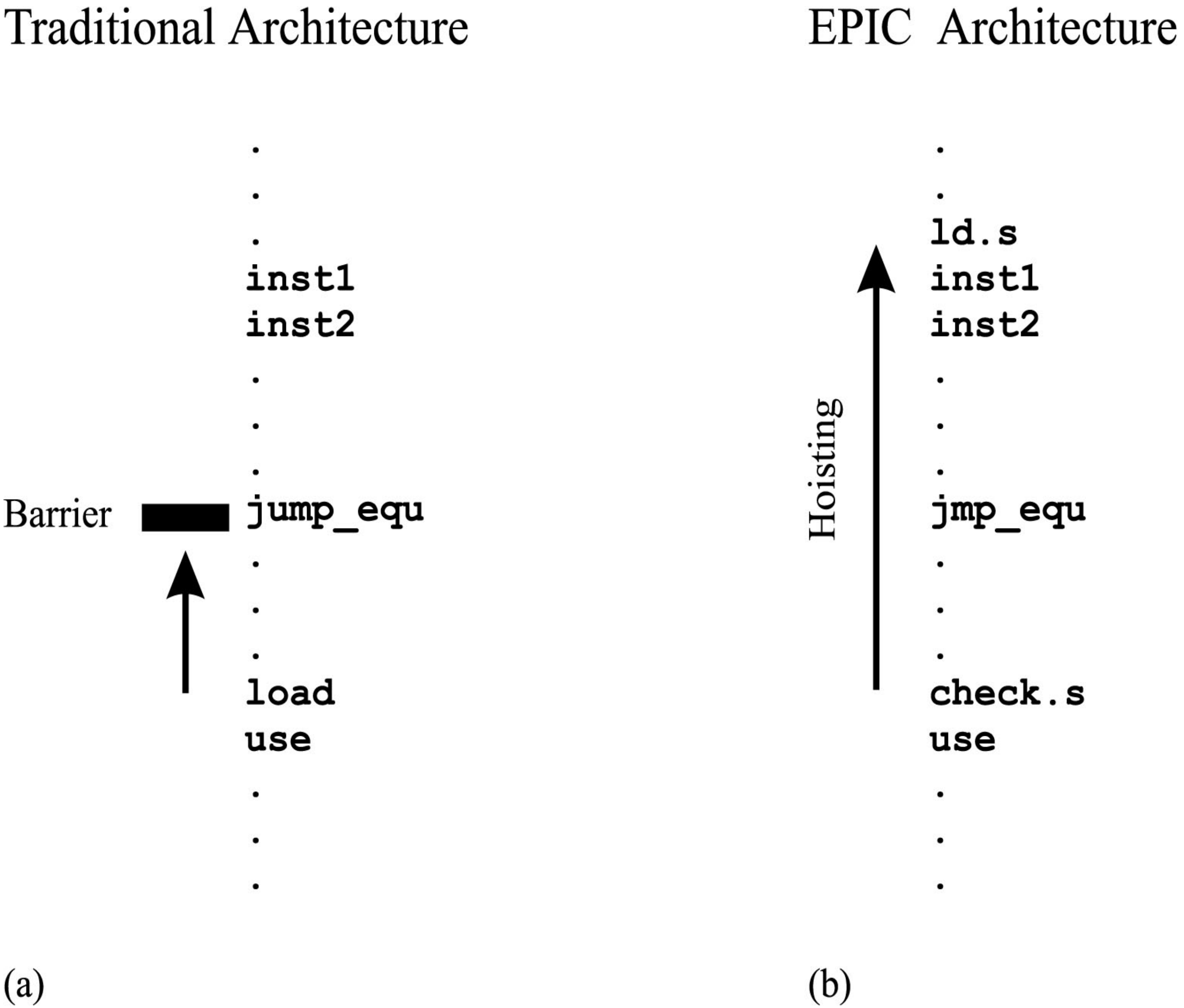
- In spite of the fact that the processor has probably executed some instructions from both possible paths, none of the (possible) results is stored yet.

- To do this, the processor checks predicate register of each of these instructions.
  - If the predicate register contains a 1,
    ===> the instruction is on the TRUE path (i.e., valid path),
    so the processor retires the instruction and stores the result.
  - If the register contains a 0,
  - ===> the instruction is invalid, so the processor discards the result.

Speculative loading

- Load data from memory well before the program needs it, and thus to effectively minimize the impact of memory latency.
- Speculative loading is a combination of compile-time and run-time optimizations. ==> compiler-controlled speculation
- The compiler is looking for any instructions that will need data from memory and, whenever possible, hoists a load at an earlier point in the instruction stream, ahead of the instruction that will actually use the data.
- Today's superscalar processors:
  - load can be hoisted up to the first branch instruction which represents a barrier
- Speculative loading combined with predication gives the compiler more flexibility to reorder instructions and to shift loads above branches.

Speculative loading - "control speculation"



Traditional Architecture

```
.
.
.
inst1
inst2
.
.
.
jump_equ
.
.
.
load
use
.
.
.
```

(a)

EPIC Architecture

```
.
.
ld.s
inst1
inst2
.
.
.
jmp_equ
.
.
.
check.s
use
.
.
.
```

(b)

Speculative loading

speculative load instruction `ld.s`

speculative check instruction `chk.s`

- The compiler:
  - inserts the matching check immediately before the particular instruction that will use the data,
  - rearranges the surrounding instructions so that the processor can issue them in parallel.
- At run-time:
  - the processor encounters the `ld.s` instruction first and tries to retrieve the data from the memory.
  - `ld.s` performs memory fetch and exception detection (e.g., checks the validity of the address).
  - If an exception is detected, `ld.s` does not deliver the exception.
  - Instead, `ld.s` only marks the target register (by setting a token bit)

Speculative loading “data speculation”

- Mechanism can also be used to move a load above a store
  even if is is not known whether the load and the store reference overlapping memory locations.

```
Ld.a  advanced load
    ...
Chk.a  check
use  data
```

Speculative loading/checking

- Exception delivery is the responsibility of the matching `chk.s` instruction.
  - When encountered, `chk.s` calls the operating system routine if the target register is marked (i.e, if the corresponding token bit is set), and does nothing otherwise.

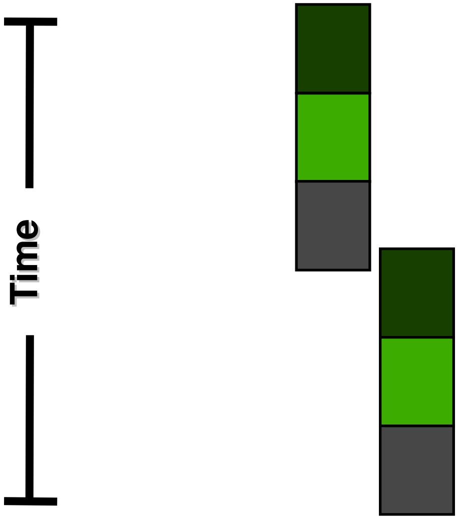
- Whether the `chk.s` instruction will be encountered may depend on the outcome of the branch instruction.

  ==> Thus, it may happen that an exception detected by `ld.s` is never delivered.

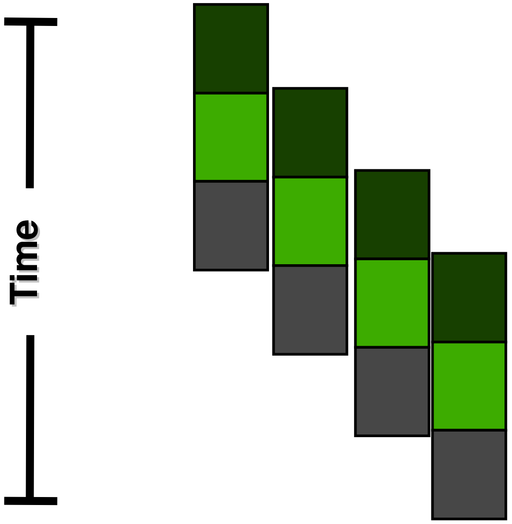
- Speculative loading with `ld.s`/`chk.s` machine level instructions resembles the `TRY`/`CATCH` statements in some high-level programming languages (e.g., Java).

Software Pipelining via Rotating Registers

- Software pipelining - improves performance by overlapping execution of different software loops - execute more loops in the same amount of time



Sequential Loop Execution



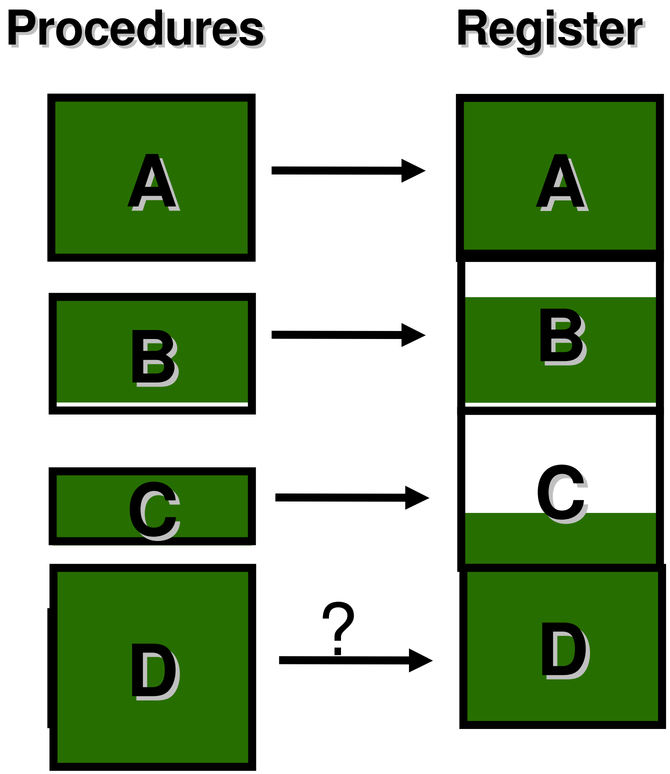
Software Pipelining Loop Execution

- Traditional architectures need complex software loop unrolling for pipelining
  - Results in code expansion --> Increases cache misses --> Reduces performance
- IA-64 utilizes rotating registers to achieve software pipelining
  - Avoids code expansion --> Reduces cache misses --> Higher performance

IA-64 Register Stack

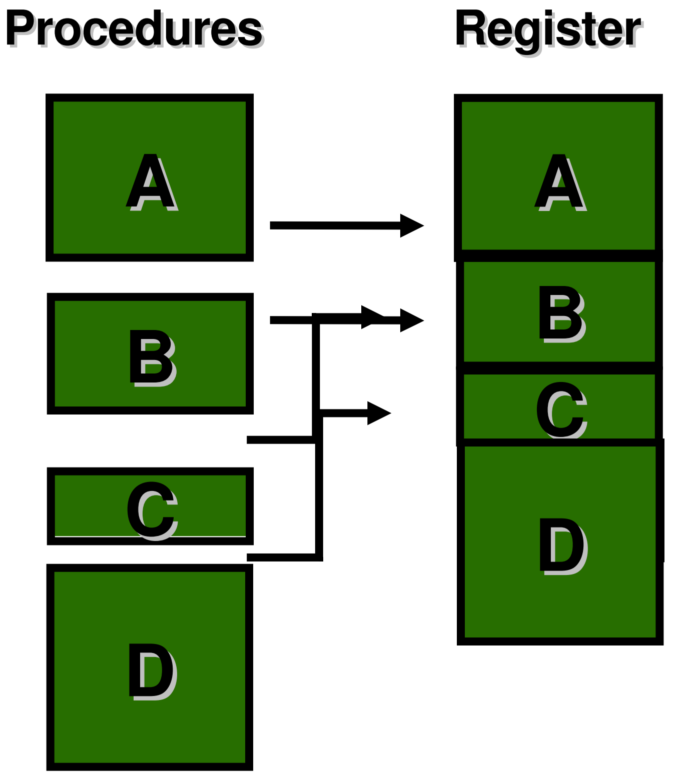
(Mulder/ Hack slide)

Traditional Register Stacks



- Eliminate the need for save / restore by reserving fixed blocks in register
- However, fixed blocks waste resources

IA-64 Register Stack



- IA-64 able to reserve variable block sizes
- No wasted resources

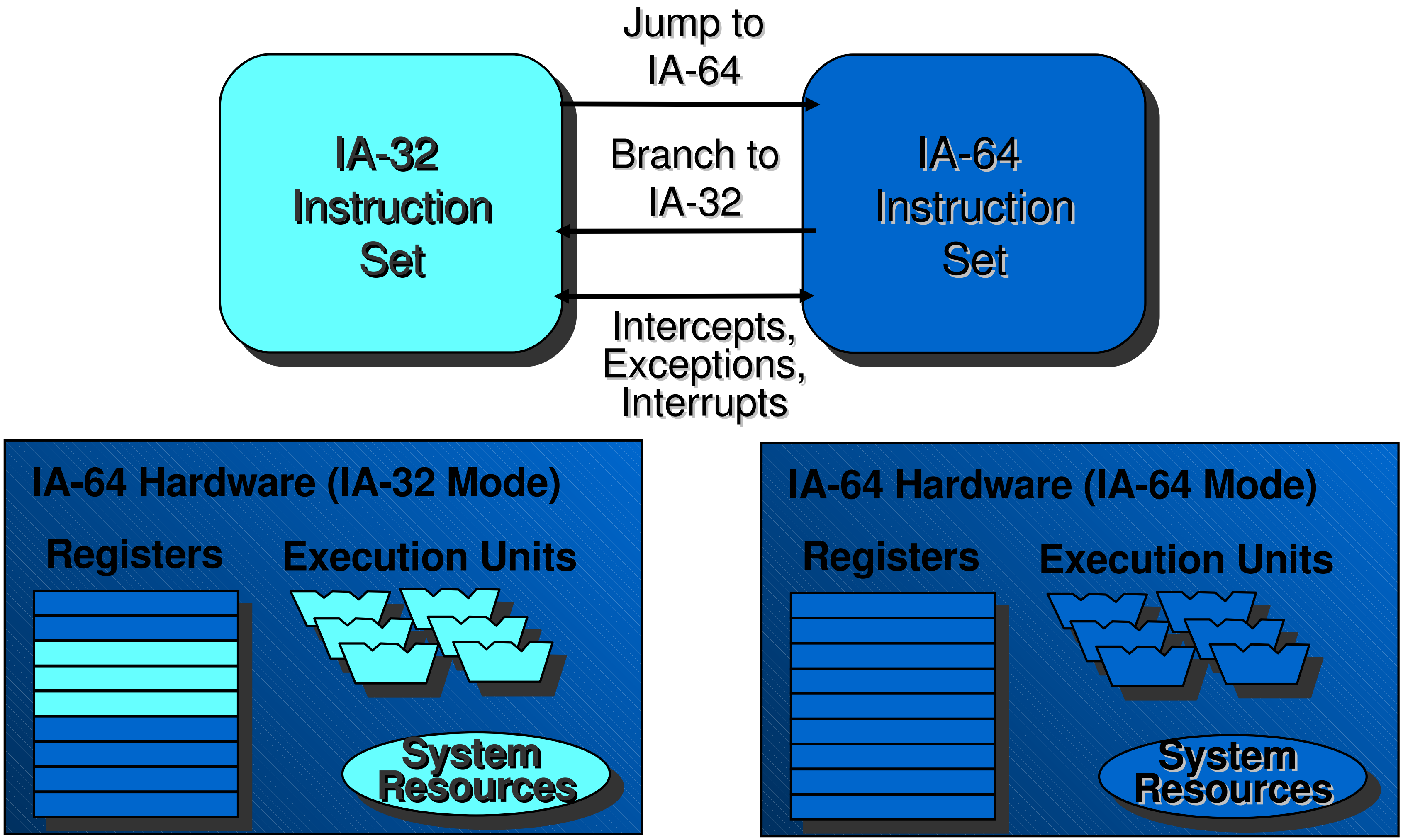
IA-64 support for Procedure Calls

- Subset of general registers are organized as a logically infinite set of stack frames that are allocated from a finite pool of physical registers

- Stacked registers are GR32 up to a user-configurable maximum of GR127
- a called procedure specifies the size of its new stack frame using `alloc` instruction
- output registers of caller are overlapped with input registers of called procedure

- Register Stack Engine:
  - management of register stack by hardware
  - moves contents of physical registers between general register file and memory
  - provides programming model that looks like unlimited register stack

Full Binary IA-32 Instruction Compatibility



Jump to IA-64

IA-32 Instruction Set

Branch to IA-32

IA-64 Instruction Set

Intercepts, Exceptions, Interrupts

IA-64 Hardware (IA-32 Mode)

Registers

Execution Units

System Resources

IA-64 Hardware (IA-64 Mode)

Registers

Execution Units

System Resources

- **IA-32 instructions supported through shared hardware resources**
- **Performance similar to volume IA-32 processors**

Full Binary Compatibility for PA-RISC

- Transparency:
  - Dynamic object code translator in HP-UX automatically converts PA-RISC code to native IA-64 code
  - Translated code is preserved for later reuse
- Correctness:
  - Has passed the same tests as the PA-8500
- Performance:
  - Close PA-RISC to IA-64 instruction mapping
  - Translation on average takes 1-2% of the time
    Native instruction execution takes 98-99%
  - Optimization done for wide instructions, predication, speculation, large register sets, etc.
  - PA-RISC optimizations carry over to IA-64

Delivery of Streaming Media

- Audio and video functions regularly perform the same operation on arrays of data values
  - IA-64 manages its resources to execute these functions efficiently
    - Able to manage general register's as 8x8, 4x16, or 2x32 bit elements
    - Multimedia operands/results reside in general registers
- IA-64 accelerates compression / decompression algorithms
  - Parallel ALU, Multiply, Shifts
  - Pack/Unpack; converts between different element sizes.
- Fully compatible with
  - IA-32 MMX™ technology,
  - Streaming SIMD Extensions and
  - PA-RISC MAX2

IA-64 3D Graphics Capabilities

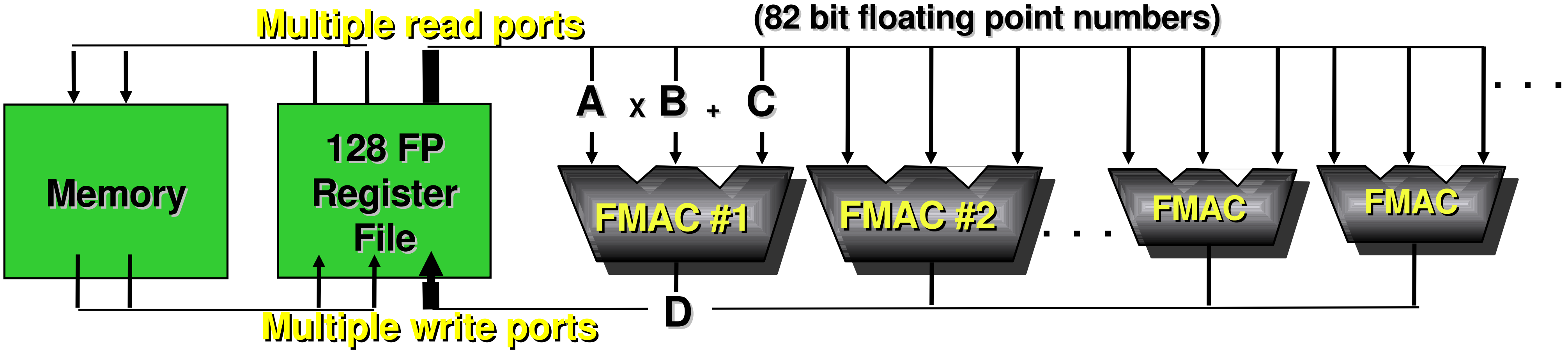
- Many geometric calculations (transforms and lighting) use 32-bit floating-point numbers
- IA-64 configures registers for maximum 32-bit floating-point performance
  - Floating-point registers treated as 2x32 bit single precision registers
  - Able to execute fast divide
  - Achieves up to 2X performance boost in 32-bit data floating-point operations
- Full support for Pentium® III processor Streaming SIMD Extensions (SSE)

* estimated

IA-64 for Scientific Analysis

- Variety of software optimizations supported
  - Load double pair : doubles bandwidth between L1 and registers
  - Full predication and speculation support
    - NaT Value to propagate deferred exceptions
    - Alternate IEEE flag sets allow preserving architectural flags
  - Software pipelining for large loop calculations
- High precision & range internal format : 82 bits
  - Mixed operations supported: single, double, extended, and 82-bit
  - Interfaces easily with memory formats
    - Simple promotion/demotion on loads/stores
  - Iterative calculations converge faster
  - Ability to handle numbers much larger than RISC competition without overflow

IA-64 Floating-Point Architecture (Mulder/ Hack slide)

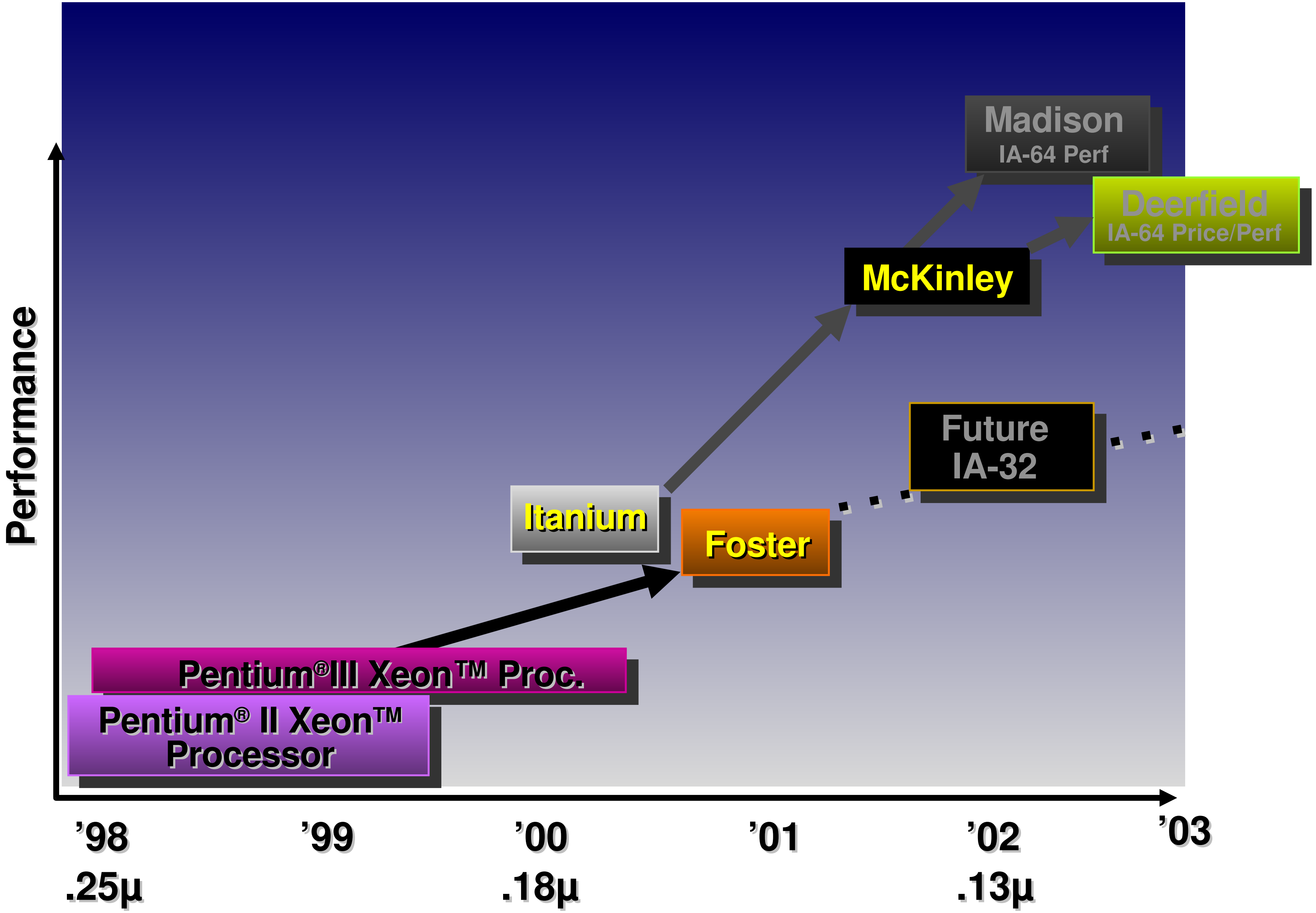


- 128 registers
  - Allows parallel execution of multiple floating-point operations
- Simultaneous Multiply - Accumulate (FMAC)
  - 3-input, 1-output operation : a * b + c = d
  - Shorter latency than independent multiply and add
  - Greater internal precision and single rounding error

Memory Support for High Performance Technical Computing

- Scientific analysis, 3D graphics and other technical workloads tend to be predictable & memory bound
- IA-64 data pre-fetching of operations allows for fast access of critical information
  - Reduces memory latency impact
- IA-64 able to specify cache allocation
  - Cache hints from load / store operations allow data to be placed at specific cache level
  - Efficient use of caches, efficient use of bandwidth

IA Server/Workstation Roadmap



Performance

Madison
IA-64 Perf

Deerfield
IA-64 Price/Perf

McKinley

Future IA-32

Itanium

Foster

Pentium®III Xeon™ Proc.

Pentium® II Xeon™ Processor

’98 .25µ

’99

’00 .18µ

’01

’02 .13µ

’03

IA-64 starts with Merced processor

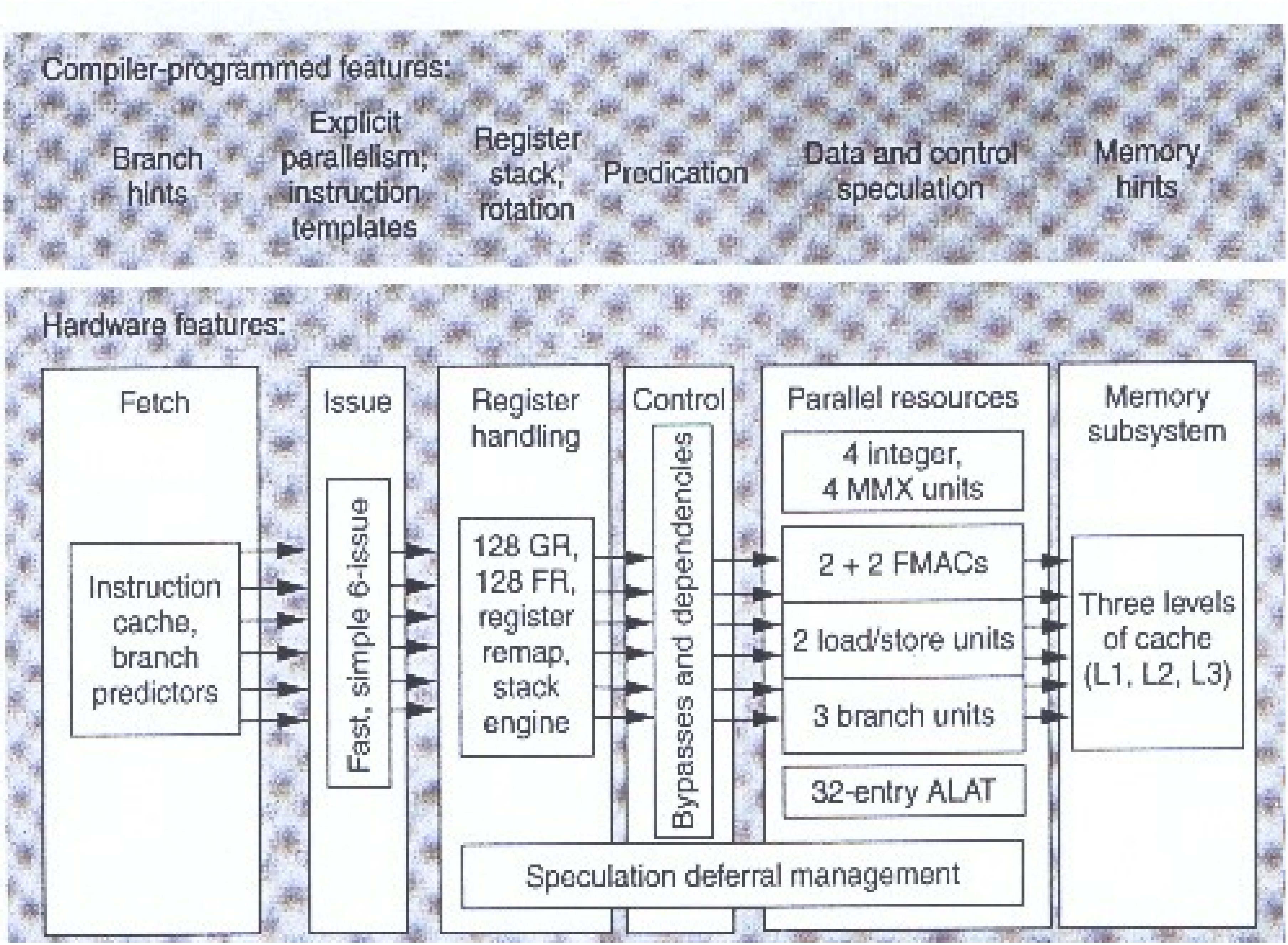
®

All dates specified are target dates provided for planning purposes only and are subject to change.

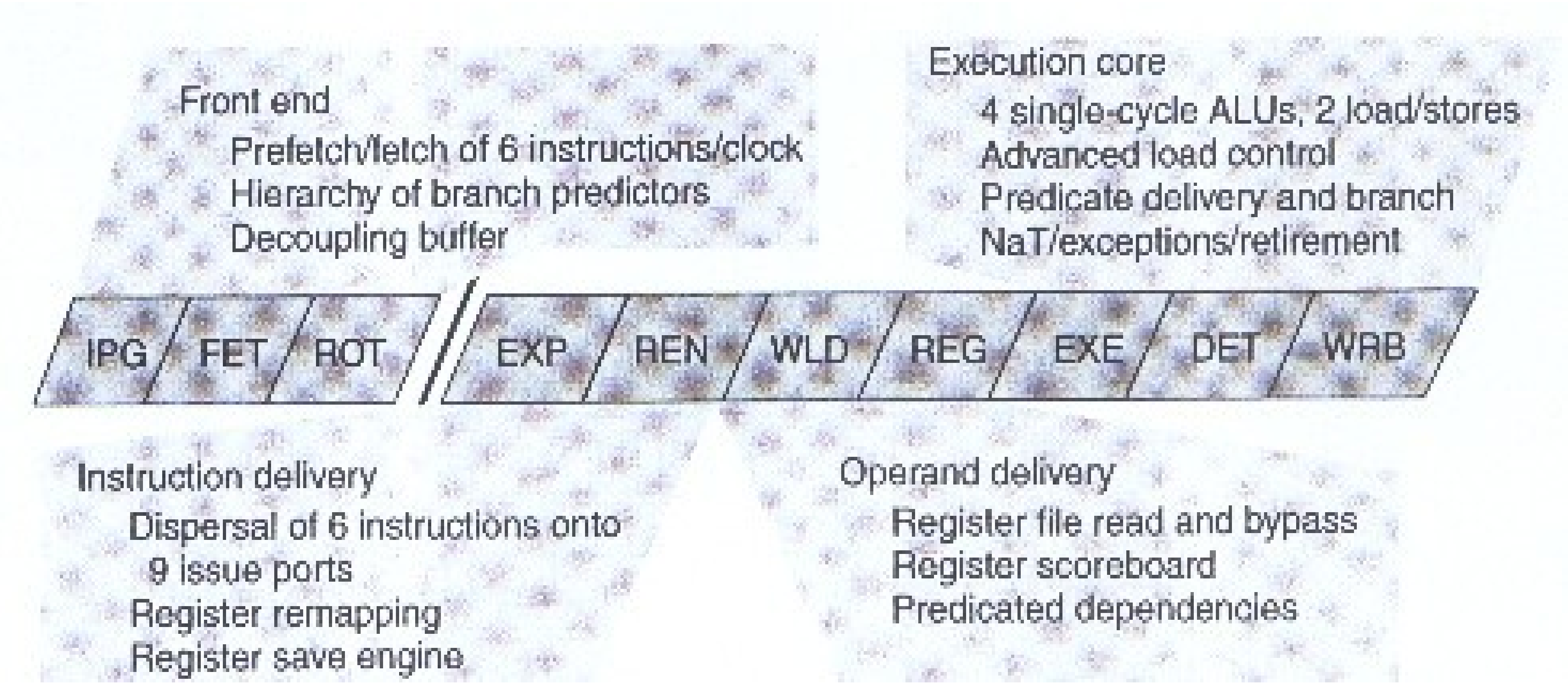
Itanium

- 64-bit processor ==> not in the Pentium, PentiumPro, Pentium II/III-line
- Targeted at servers with moderate to large numbers of processors
- full compatibility with Intel’s IA-32 ISA
- **EPIC** (*explicitly parallel instruction computing*) is applied.
- 6-wide (3 EPIC instructions) pipeline
- 10 stage pipeline
- 4 int, 4 multimedia, 2 load/store, 3 branch, 2 extended floating-point, 2 single-prec. Floating-point units
- Multi-level branch prediction besides predication
- 16 KB 4-way set-associative d- and I-caches
- 96 KB 6-way set-associative L2 cache
- 4 MB L3 cache (on package)
- 800 MHz, 0.18 micro process (at beginning of 2001)
- shipments end of 1999 or mid-2000 or ??

Conceptual View of Itanium



Itanium Processor Core Pipeline

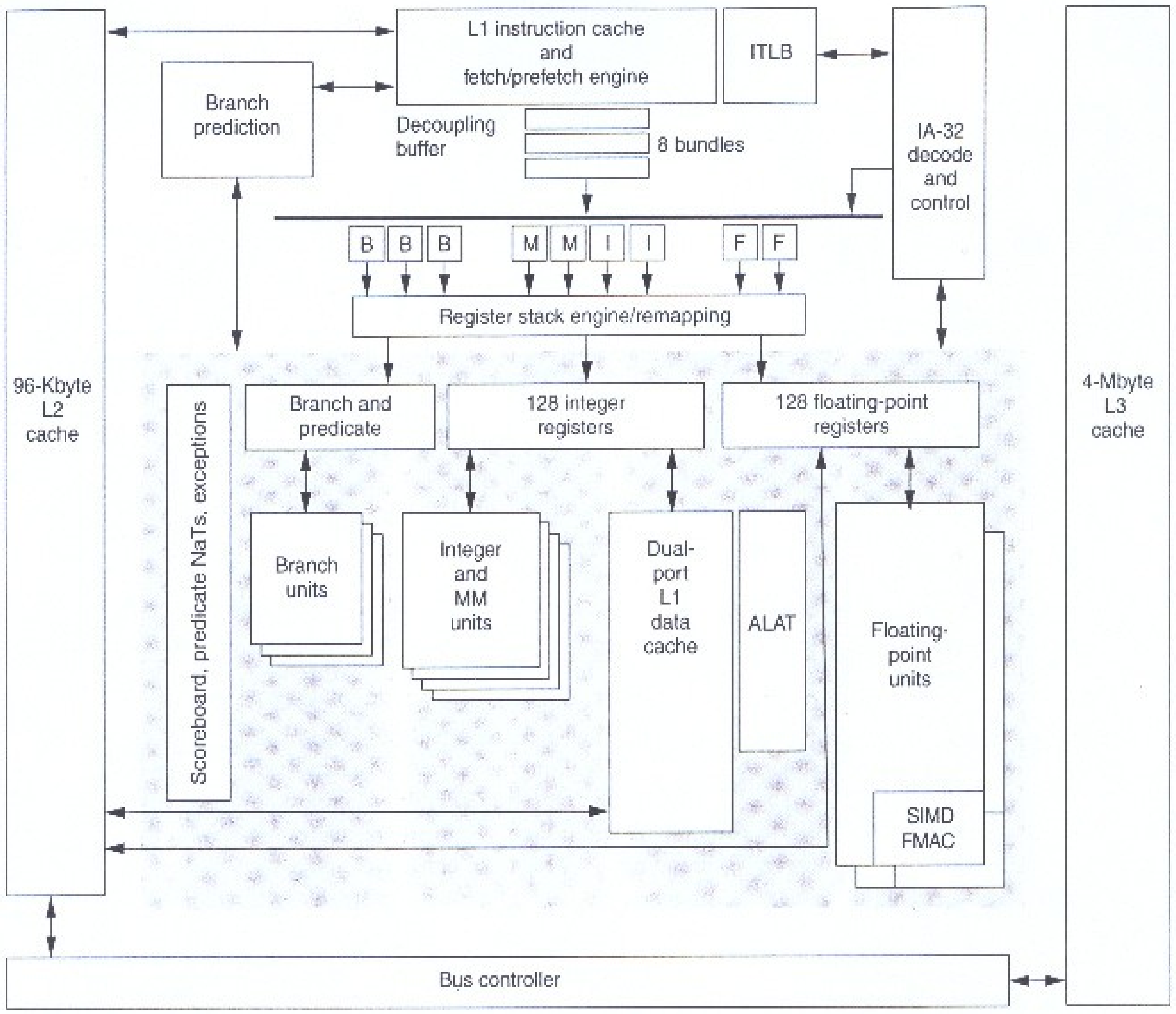


ROT: instruction rotation

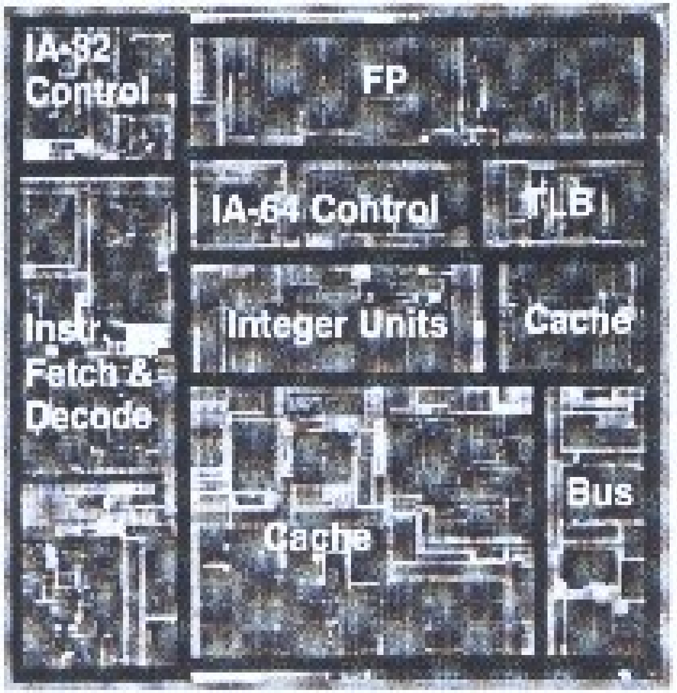
pipelined access of the large register file:

- WDL: word line decode:
- REG: register read

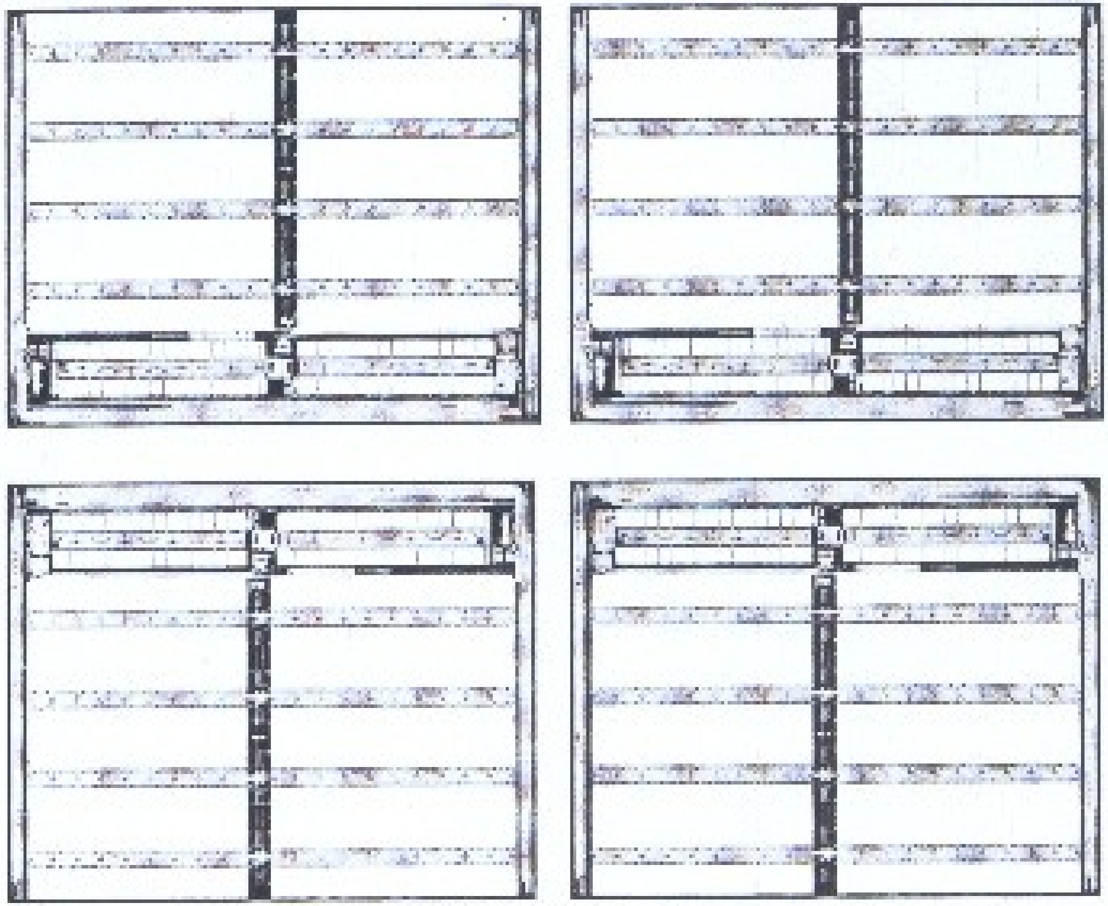
DET: exception detection (~retire stage)


L1 instruction cache
and
fetch/prefetch engine
ITLB
Branch
prediction
Decoupling
buffer
8 bundles
IA-32
decode
and
control
B
B
B
M
M
I
I
F
F
Register stack engine/remapping
96-Kbyte
L2
cache
Scoreboard, predicate NaTs, exceptions
Branch and
predicate
128 integer
registers
128 floating-point
registers
4-Mbyte
L3
cache
Branch
units
Integer
and
MM
units
Dual-
port
L1
data
cache
ALAT
Floating-
point
units
SIMD
FMAC
Bus controller

Itanium Die Plot



Core processor die



4 x 1Mbyte L3 cache

Itanium vs. Willamette (P4)

- Itanium announced with 800 MHz
- P4 announced with 1.2 GHz

⋗P4 may be faster in running IA-32 code than Itanium running IA-64 code

- Itanium probably won‘t compete with contemporary IA-32 processors
- but Intel will complete the Itanium design anyway

- Intel hopes for the Itanium successor **McKinley** which will be out only one year later