The EPIC-VLIW Approach

- Explicitly Parallel Instruction Computing (EPIC) is a "philosophy"
- Very Long Instruction Word (VLIW) is an implementation of EPIC
- Concept derives from horizontal microprogramming, namely:
  - A sequence of steps (microoperation) that interprets the ISA
  - If only one microop per cycle: vertical microprogramming
  - If (at the extreme all) several units (say, incr PC, add, f-p, register file read, register file write etc…) can be activated in the same cycle: horizontal microprogramming

The EPIC "philosophy"

- Compiler generates packets, or bundles, of instructions that can execute together
  - Instructions executed in order (static scheduling) and assumed to have a fixed latency
- Architecture should provide features that assists the compiler in exploiting ILP
  - Branch prediction, load speculation (see later), and associated recoveries
- Difficulties occur with unpredictable latencies :
  - Branch prediction → Use of predication in addition to static and dynamic branch prediction
  - Pointer-based computations →Use cache hints, speculative loads

Why EPIC?

- Dynamically scheduled processors have (lower CPI) better performance than statically scheduled ones. So why EPIC?
- Statically scheduled hardware is simpler
  - Examples?
- Static scheduling can look at the whole program rather than a relatively small instruction window. More possibilities of optimization
  - R1 ← R2 + R3 latency 1
  - R3 ← R4 * R5 latency m; could start m-2 cycles before the addition

Other Static Scheduling Techniques

- Eliminate branches via predication (next slides)
- Loop unrolling
- Software pipelining (see in a few slides)
- Use of global scheduling
  - Trace scheduling technique: focus on the critical path
- Software prefetching
  - We'll talk about prefetching at length later

Predication Basic Idea

- Associate a Boolean condition (predicate) with the issue, execution, or commit of an instruction
  - The stage in which to test the predicate is an implementation choice
- If the predicate is true, the result of the instruction is kept
- If the predicate is false, the instruction is nullified
- Distinction between
  - Partial predication: only a few opcodes can be predicated
  - Full predication: every instruction is predicated

Predication Benefits

- Allows compiler to overlap the execution of independent control constructs w/o code explosion
- Allows compiler to reduce frequency of branch instructions and, consequently, of branch mispredictions
- Reduces the number of branches to be tested in a given cycle
- Reduces the number of multiple execution paths and associated hardware costs (copies of register maps etc.)
- Allows code movement in superblocks

Predication Costs

- Increased fetch utilization
- Increased register consumption
- If predication is tested at commit time, increased functional-unit utilization
- With code movement, increased complexity of exception handling
  - For example, insert extra instructions for exception checking
- If every instruction is predicated, larger instruction
  - Impacts I-cache

Flavors of Predication Implementation

- Has its roots in vector machines like CRAY-1
  - Creation of vector masks to control vector operations on an element per element basis
- Often (partial) predication limited to conditional moves as, e.g., in the Alpha, MIPS 10000, IBM Power PC, SPARC and Intel P6 microarchitecture
- Full predication: Every instruction predicated as in Intel Itanium (IA-64 ISA)

Partial Predication: Conditional Moves

- CMOV R1, R2, R3
  - Move R2 to R1 if R3 $\neq$ 0
- Main compiler use: If (cond) S1 (with result in Rres)
  - (1) Compute result of S1 in Rs1;
  - (2) Compute condition in Rcond;
  - (3) CMOV Rres, Rs1, Rcond
- No need (in this example) for branch prediction
- Very useful if condition can be computed ahead or, e.g., in parallel with result.
- But**:** Increases register pressure (Rcond is general register)

Other Forms of Partial Predication

- Select dest, src1, src2,cond
  - Corresponds to C-like --- *dest = ((cond) ? src1 : src2)*
  - Note the destination register is always assigned a value
  - Use in the Multiflow (first commercial VLIW machine)
- Nullify
  - Any register-register instruction can nullify the next instruction, thus making it conditional

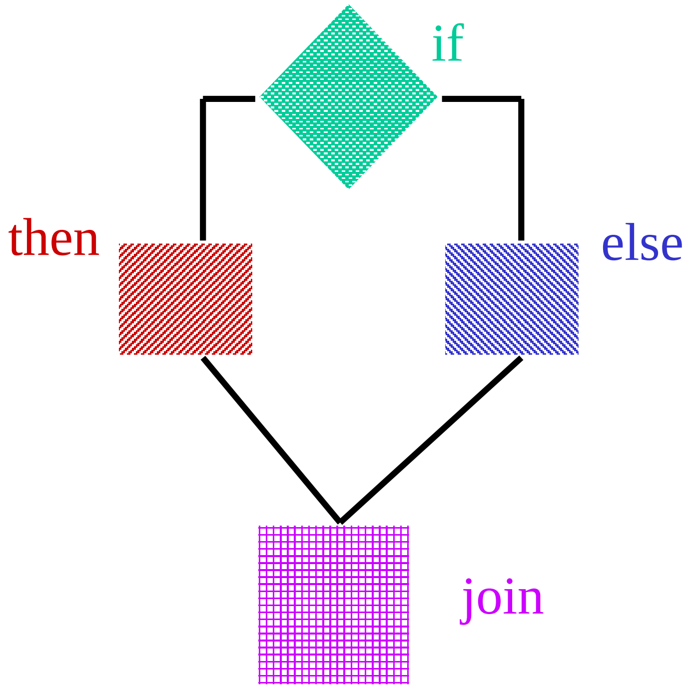
Full Predication

- Define predicates with instructions of the form:

  Pred_<*cmp*> Pout1$_{<type>}$, Pout2$_{<type>}$,, src1, src2 (P_{in}) where

  - Pout1 and Pout2 are assigned values according to the comparison between src1 and src2 and the cmp “opcode”
  - The predicate types are most often U (unconditional) and $\overline{U}$ its complement, and OR and $\overline{OR}$
  - The predicate define instruction can itself be predicated with the value of P_{in}
    - There are definite rules for that, e.g., if P_{in} = 0, U and $\overline{U}$ are set to 0 independently of the result of the comparison and the OR predicates are not modified.

If-conversion



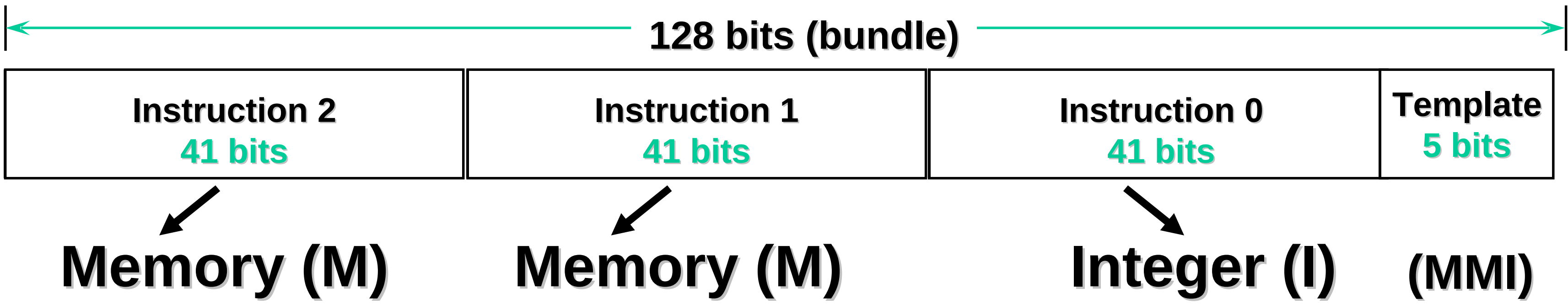
The if condition will set p1 to U

The then will be executed predicated on p1(U)

The else will be executed predicated on p1($\overline{U}$)

The “join” will in general be predicated on some form of OR predicate

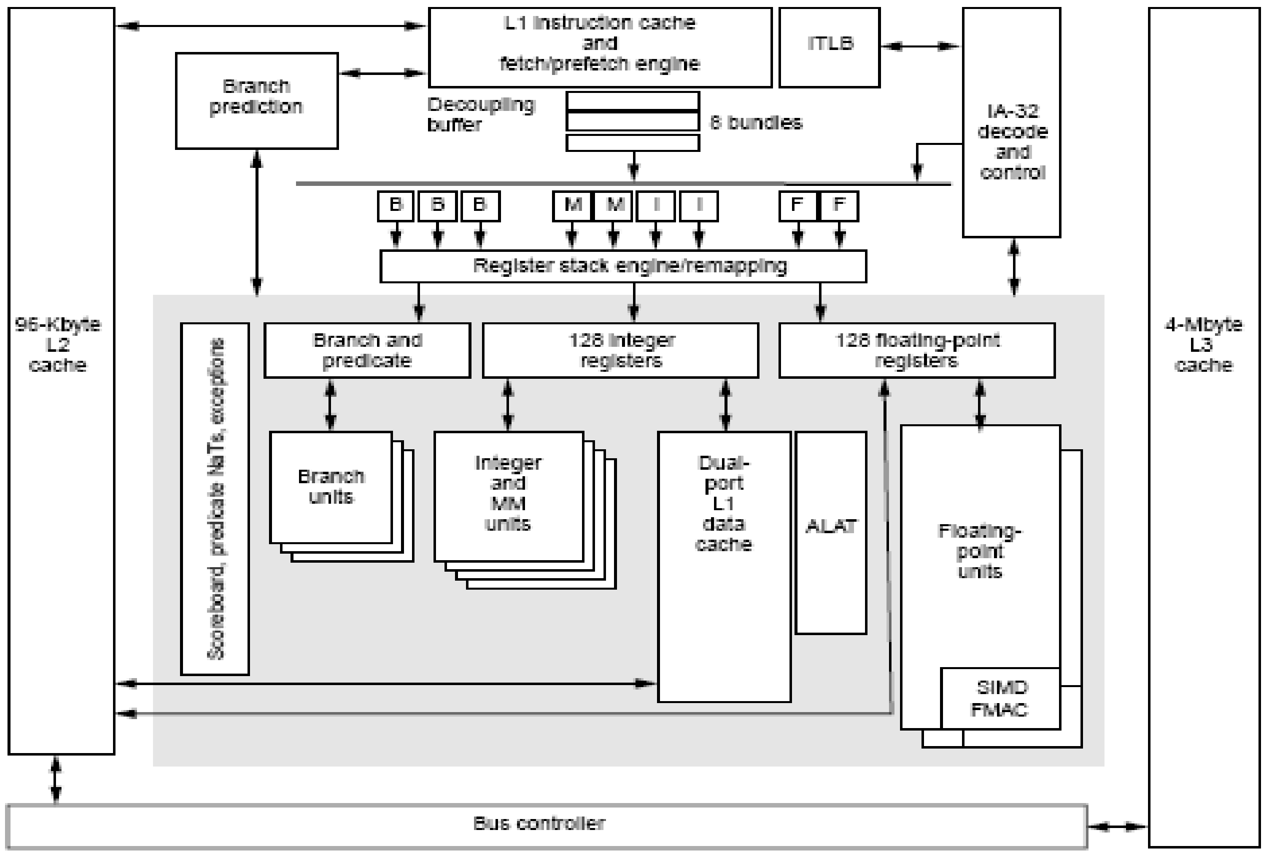
IA-64 : Explicitly Parallel Architecture



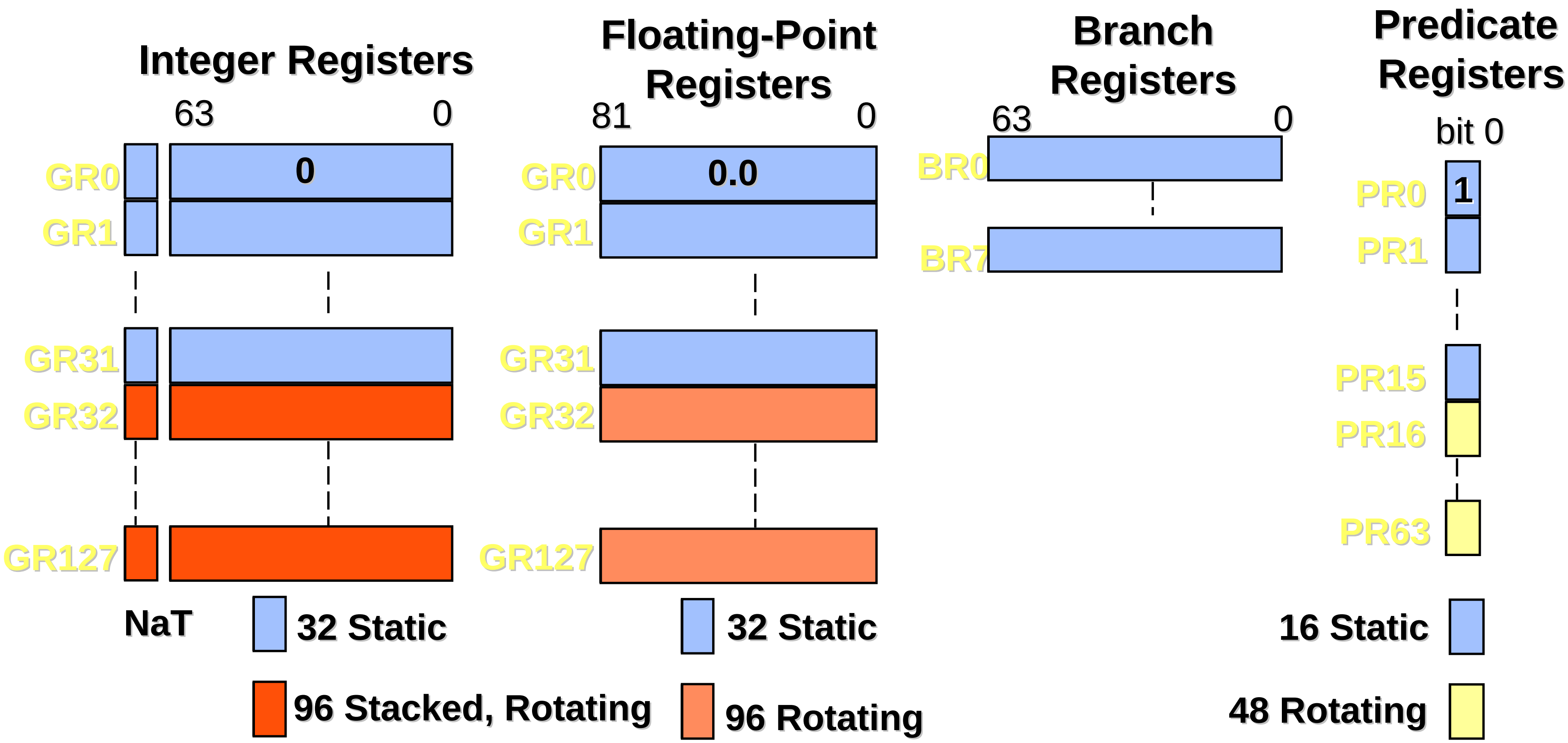
- IA-64 template specifies
  - The type of operation for each instruction, e.g.
    - MFI, MMI, MII, MLI, MIB, MMF, MFB, MMB, MBB, BBB
  - Intra-bundle relationship, e.g.
    - M / MI or MI / I (/ is a "stop" meaning no parallelism)
  - Inter-bundle relationship
- Most common combinations covered by 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

Itanium Overview



IA-64's Large Register File



Itanium implementation

- Can execute 2 bundles (6 instructions) per cycle
- 10 stage pipeline
- 4 integer units (2 of them can handle load-store), 2 f-p units and 3 branch units
- Issue in order, execute in order but can complete out of order. Uses a (restricted) register scoreboard technique to resolve dependencies.

Itanium implementation

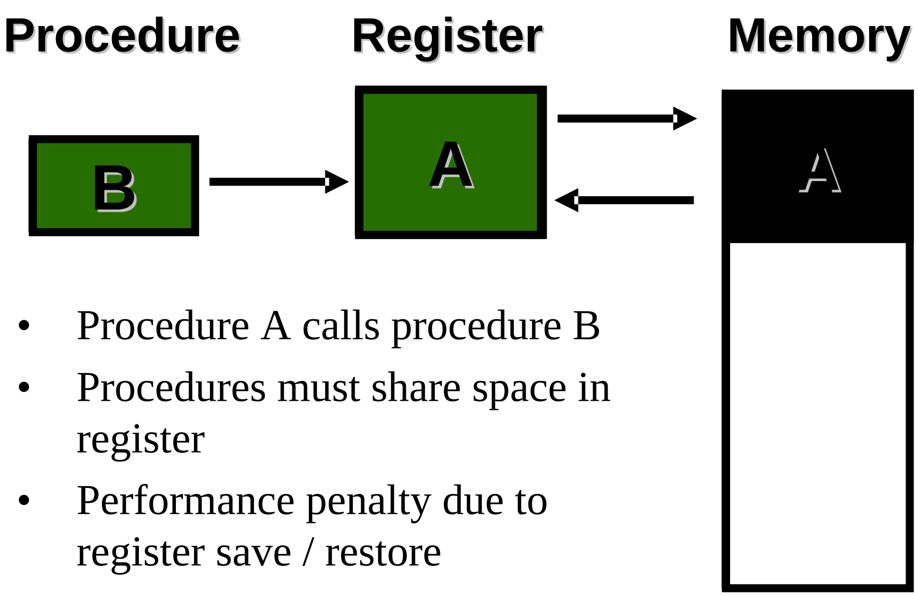
- Predication reduces number of branches and number of mispredicts,
- Nonetheless: sophisticated branch predictor
  - Two level branch predictor of the SAs variety
  - Some provision for multiway branches
    - Several basic blocks can terminate in the same bundle
  - 4 registers for highly predictable target addresses (end of loops) hence no bubble on taken branch
  - Return address stack
  - Hints from the compiler
  - Possibility of prefetching instructions from L2 to instruction buffer

Itanium implementation

- There are "instruction queues" between the fetch unit and the execution units. Therefore branch bubbles can often be absorbed because of long latencies (and stalls) in the execute stages
- Some form of scoreboarding is used for detecting dependencies

Traditional Register Models

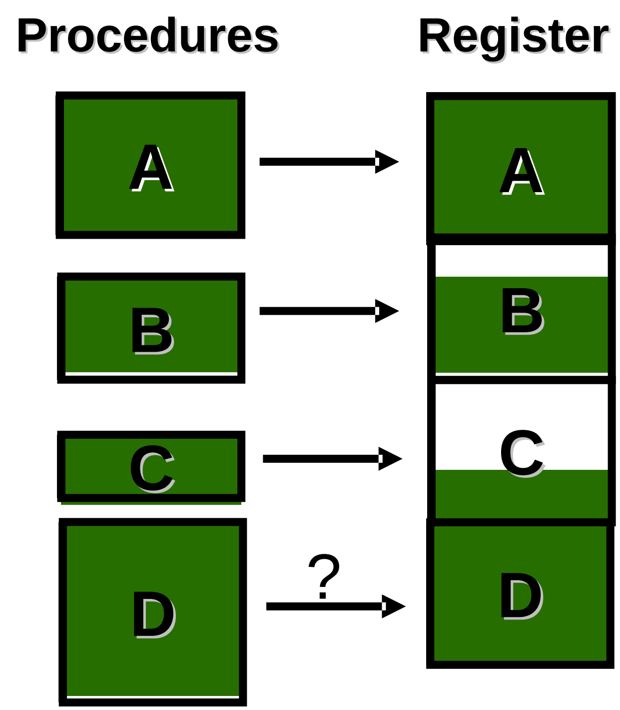
Traditional Register Models



- Procedure A calls procedure B
- Procedures must share space in register
- Performance penalty due to register save / restore

I think that the “traditional register stack” model they refer to is the “register windows” model used in Sparc

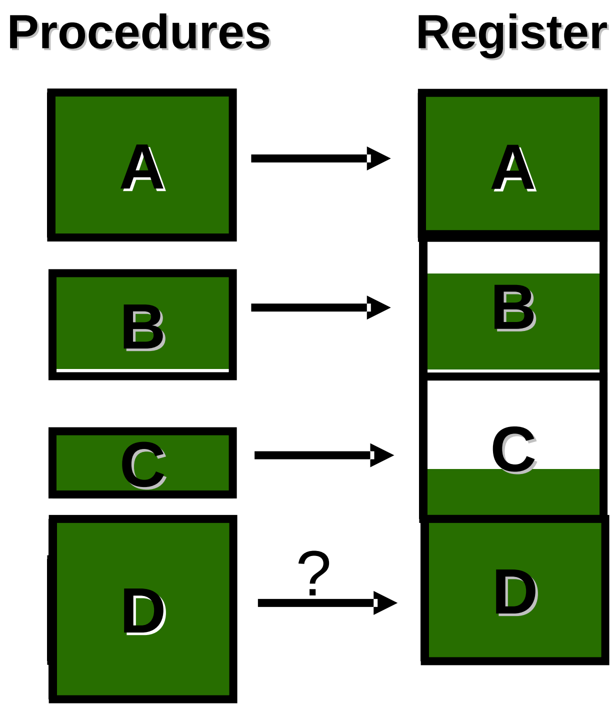
Traditional Register Stacks



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

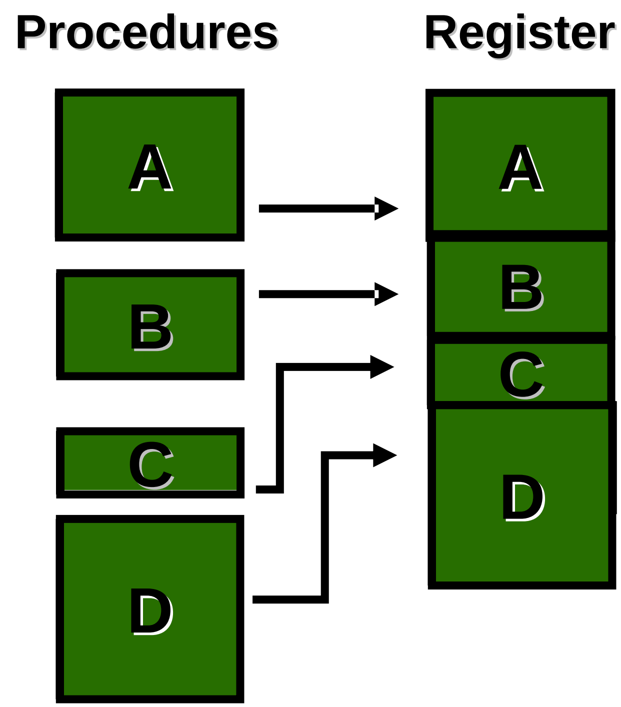
IA-64 Register Stack

Traditional Register Stacks



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

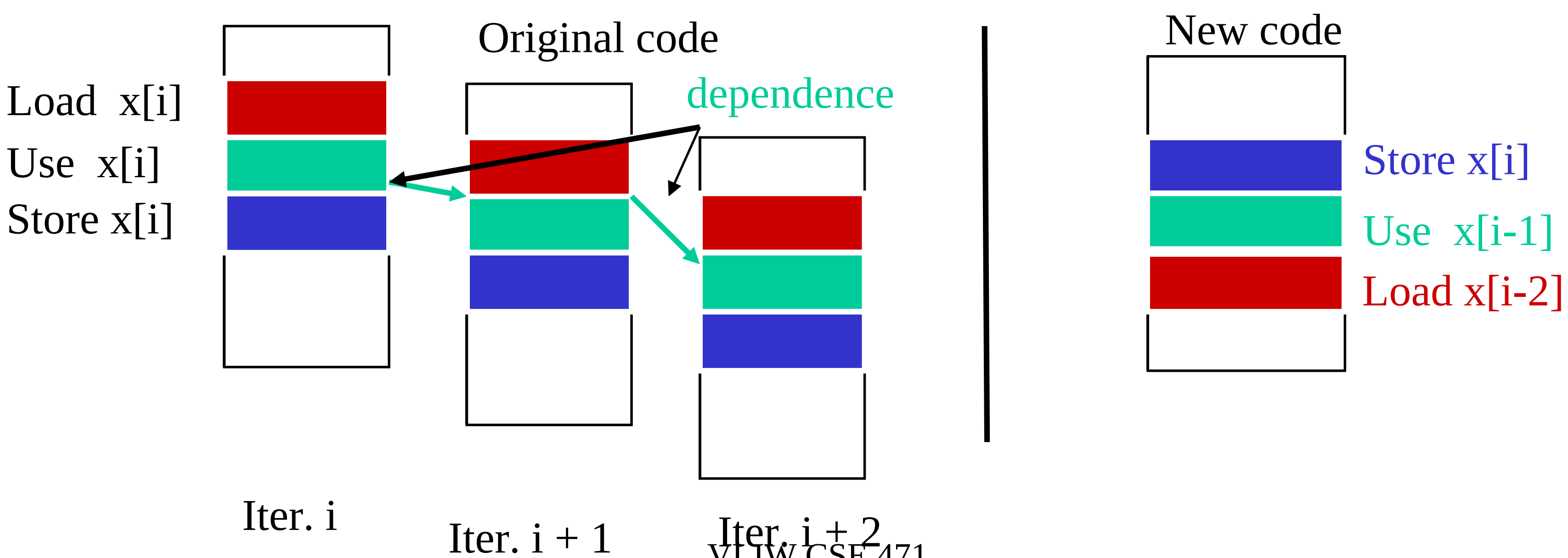
IA-64 Register Stack



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

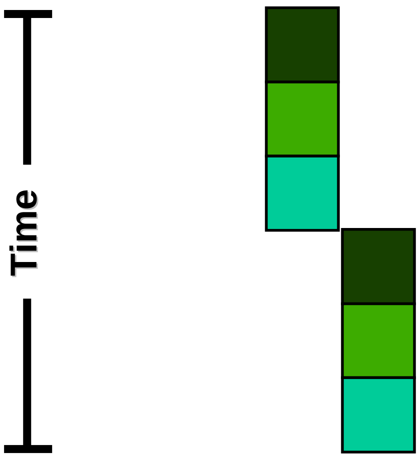
Software pipelining

- Reorganize loops with loop-carried dependences by "symbolically" unrolling them
  - New code : statements of distinct iterations of original code
  - Take an "horizontal" slice of several (dependent) iterations

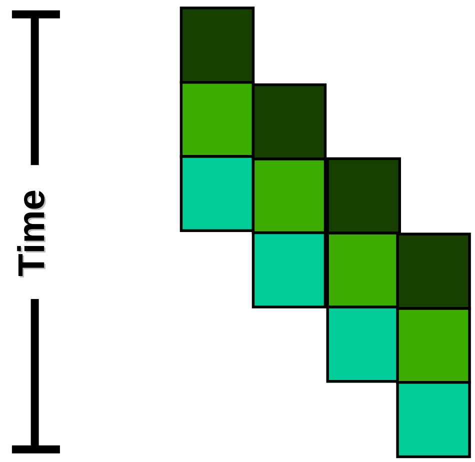


Software Pipelining via Rotating Registers

Sequential Loop Execution

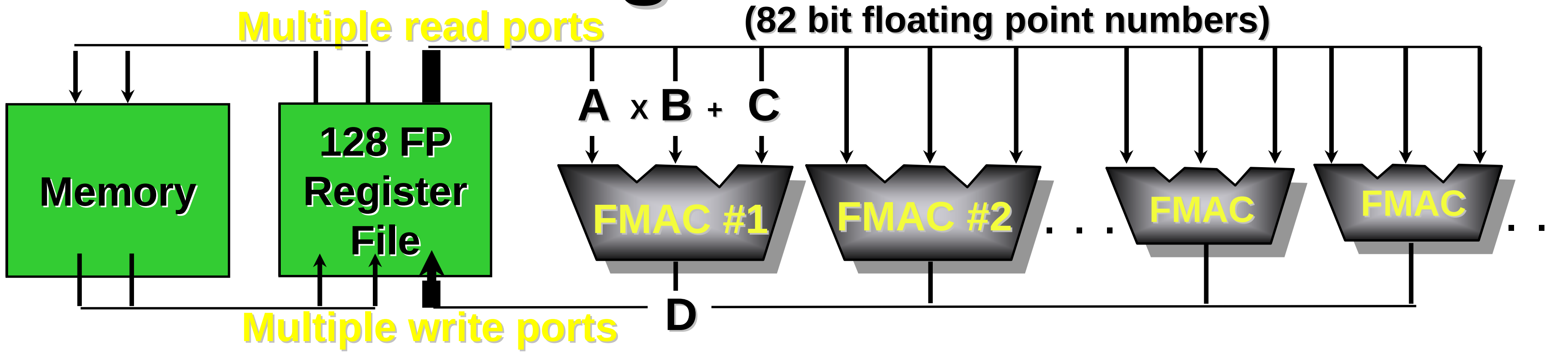


Software Pipelining Loop Execution



- Traditional architectures need complex software loop unrolling for pipelining
  - Results in code expansion --> Increases cache misses --> Reduces performance
- IA-64 utilizes rotating registers (r0 ->r1, r1 -> r2 etc in successive iterations) to achieve software pipelining
  - Avoids code expansion --> Reduces cache misses --> Higher performance

IA-64 Floating-Point Architecture



- 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