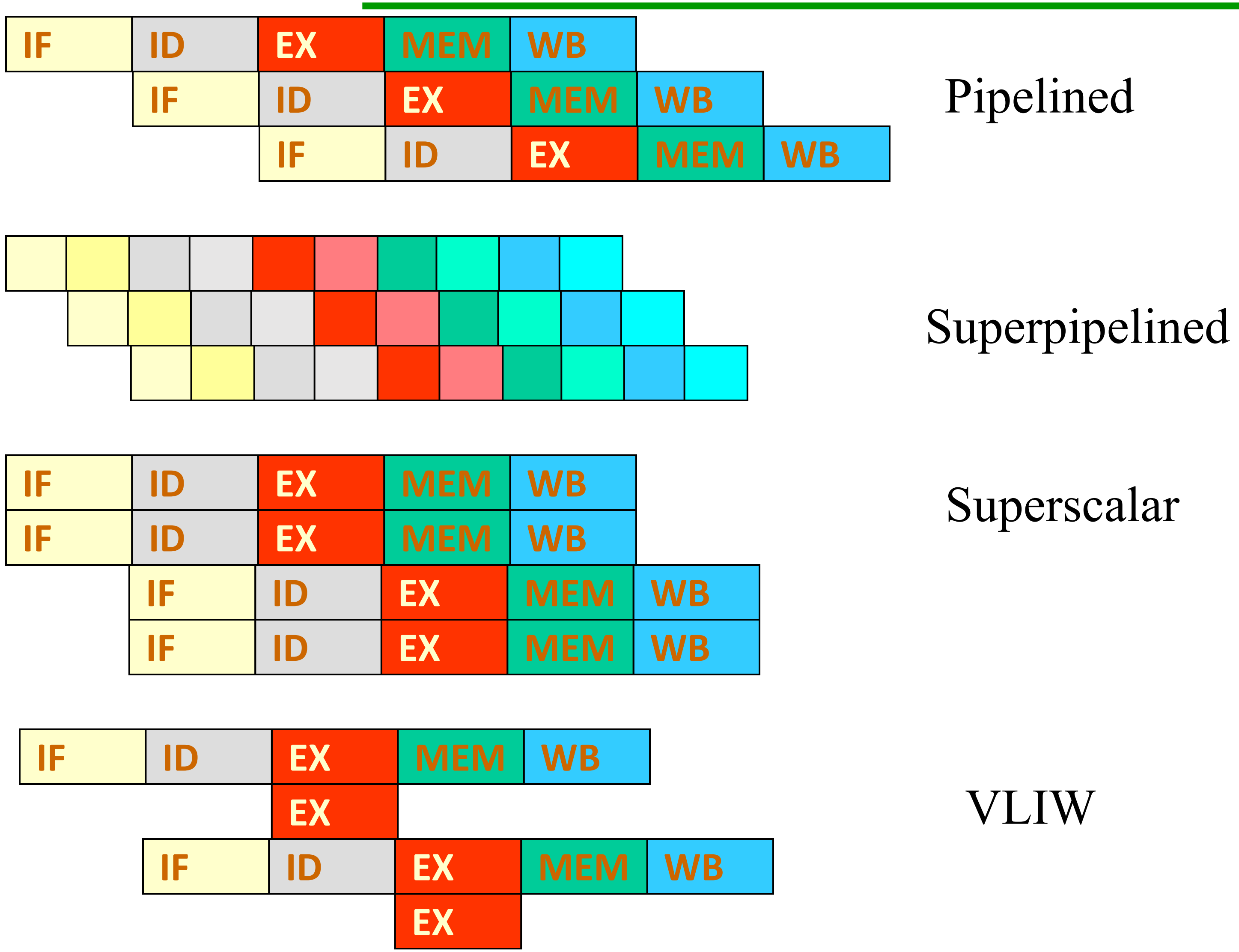
ILP Processor - ILP Architectures

- Superscalar Architecture
- VLIW Architecture
- EPIC
- Subword Parallelism, …

Motivation for ILP

- How to achieve an IPC > 1 or CPI < 1?
- Exploit Instruction Level Parallelism (ILP)
- Multiple Instruction Issue and execution
  - Superscalar and VLIW architectures
  - Run-time vs. Compile-time approach.
  - Must maintain sequential specification, although removes non-essential sequentiality and achieves parallel execution where possible.

Comparison


IF ID EX MEM WB
IF ID EX MEM WB
IF ID EX MEM WB
Pipelined
Superpipelined
IF ID EX MEM WB
IF ID EX MEM WB
IF ID EX MEM WB
IF ID EX MEM WB
Superscalar
IF ID EX MEM WB
EX
IF ID EX MEM WB
EX
VLIW

Instruction-Level Parallelism

- Very Long Instruction Word (VLIW) Architecture
  - Multiple operations per instruction packed at Compile time
  - Compiler/Programmer responsibility to expose ILP
  - Less hardware complexity

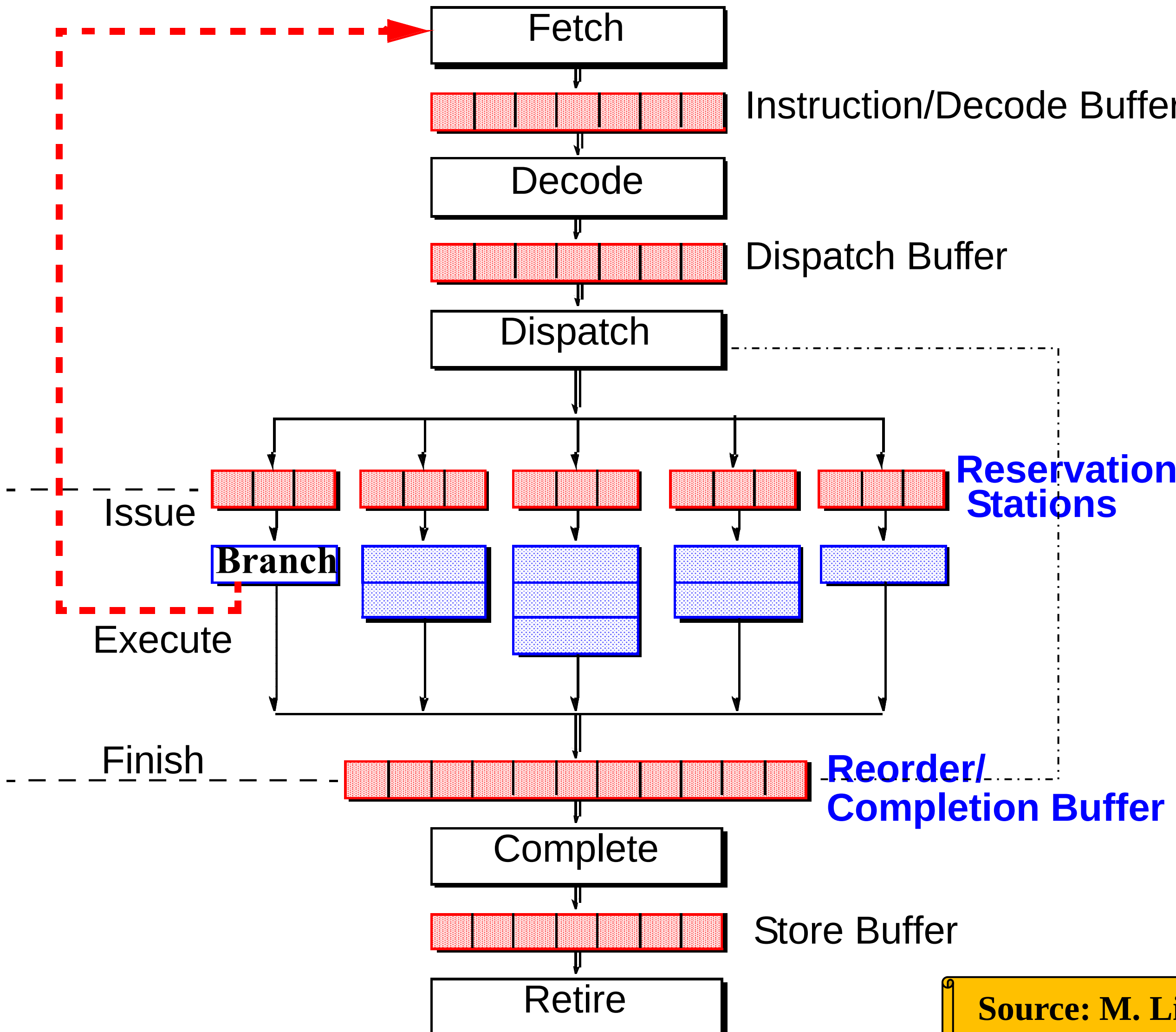
  Examples: Cydra, Multi-Flow, TI-C6x, Trimedia.
- Superpipelining: stages of the pipeline are further pipelined.
  - Results in deep pipelines and faster clocks.

Introduction (contd.)

- Superscalar Architecture
  - Multiple instructions are fetched, decoded, issued and executed each cycle.
  - Hardware detection of which instructions can be executed in parallel in a cycle at run-time.
  - *In-order* vs. *Out-of-Order* Issue

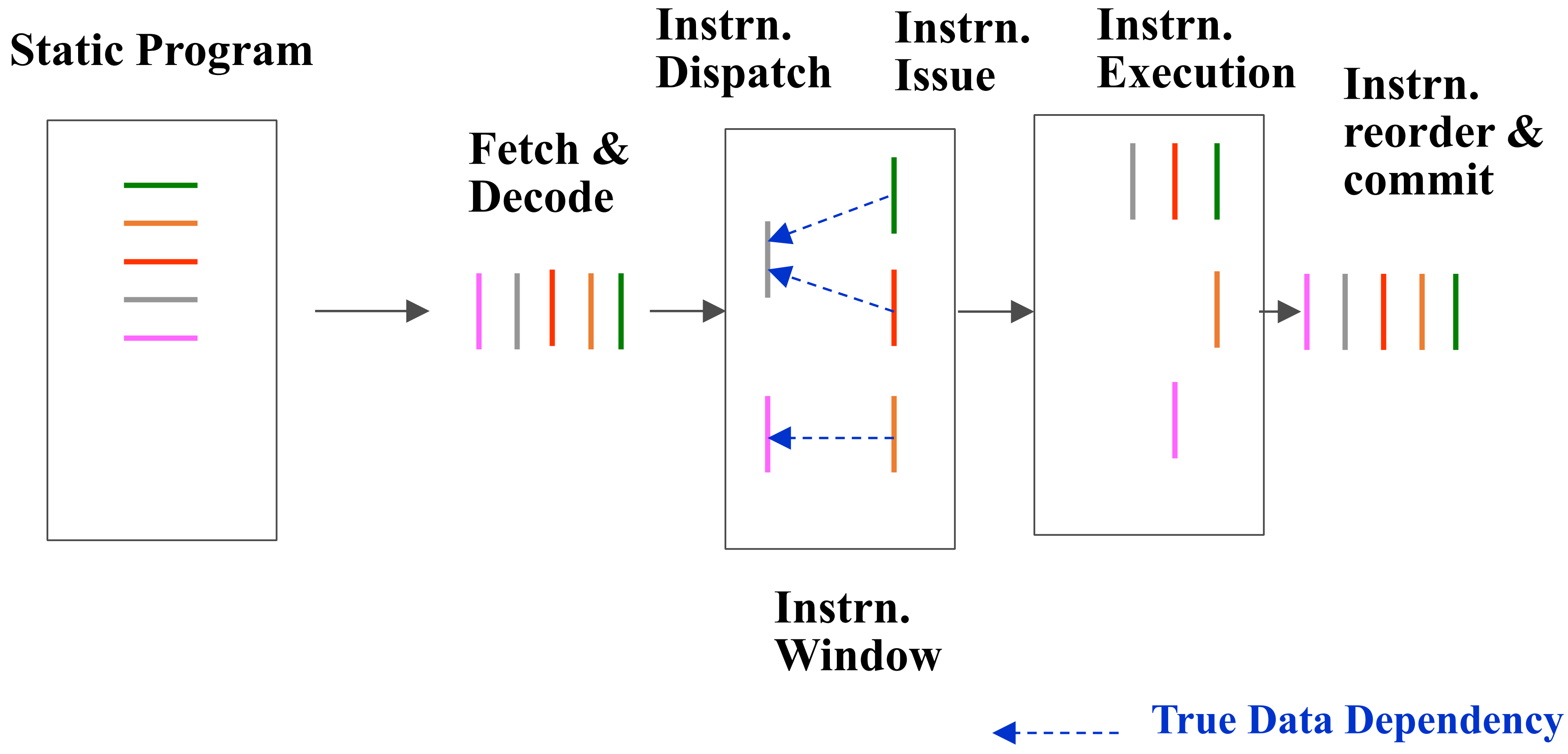
Examples: Most modern processors are superpipelined, superscalar architectures: DEC 21x64, MIPS R-8000, R-10000, PA-RISC, PowerPC-620, Pentium, UltraSparc I, II, etc.

Superscalar Pipeline Stages

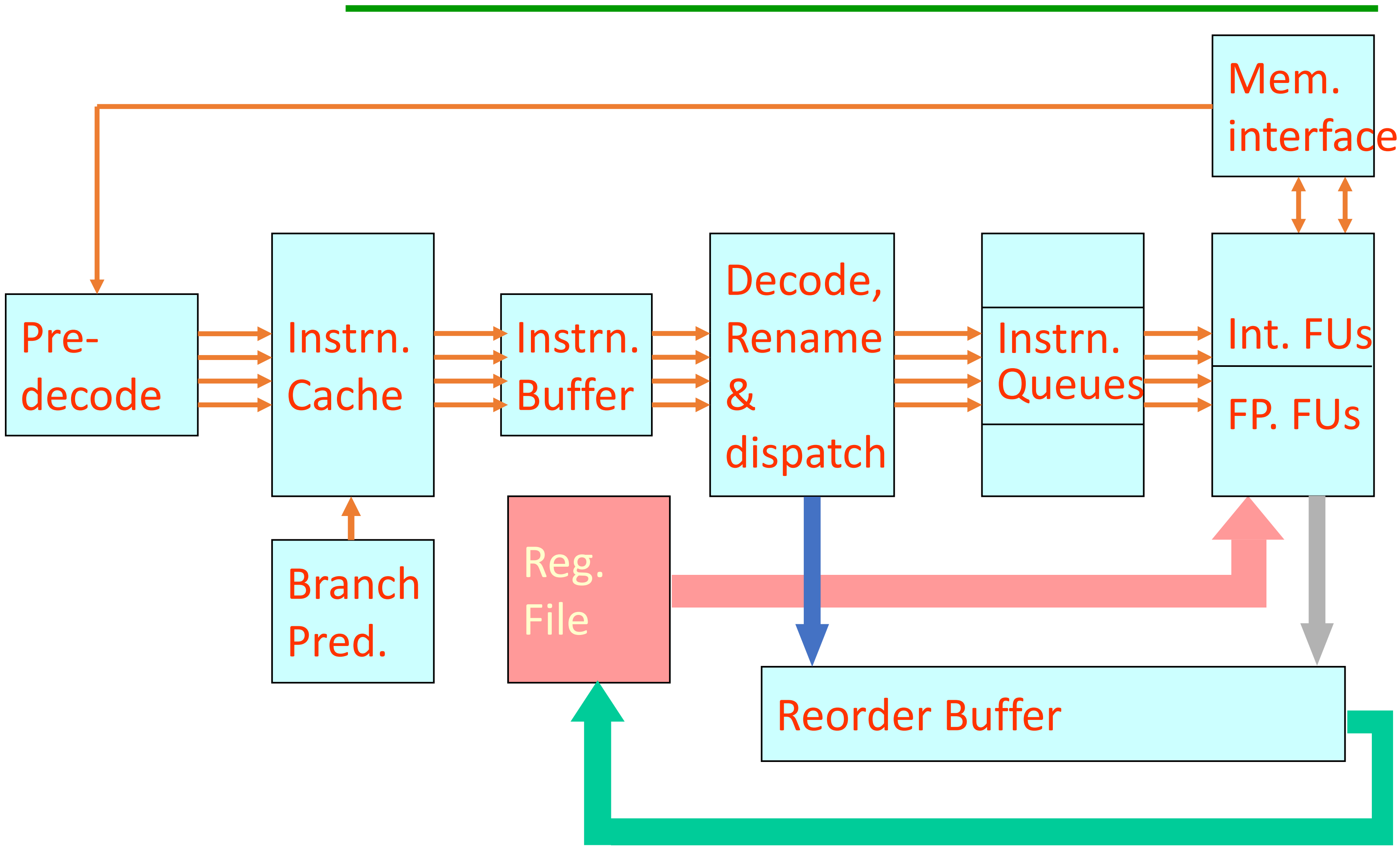


Source: M. Lipasti @ U Wisc.

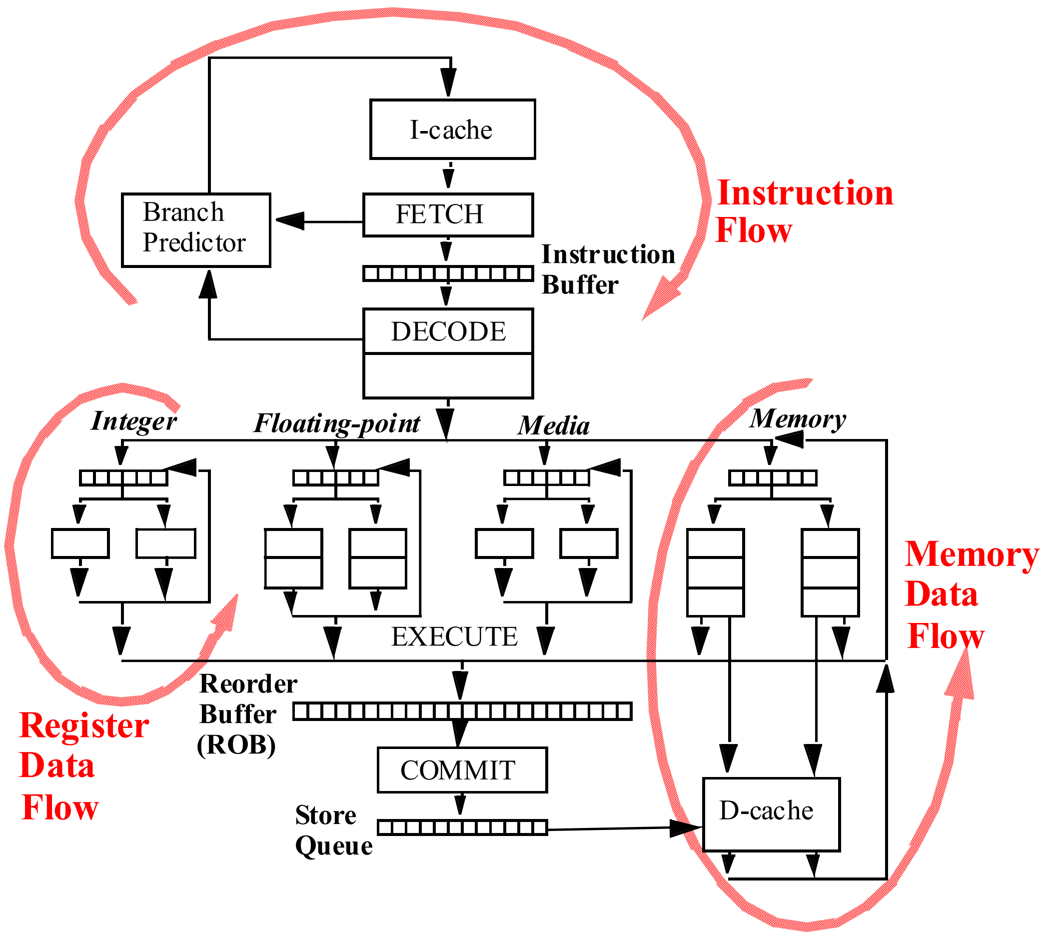
Superscalar Execution Model



Microarchitecture Overview



Impediments to High IPC

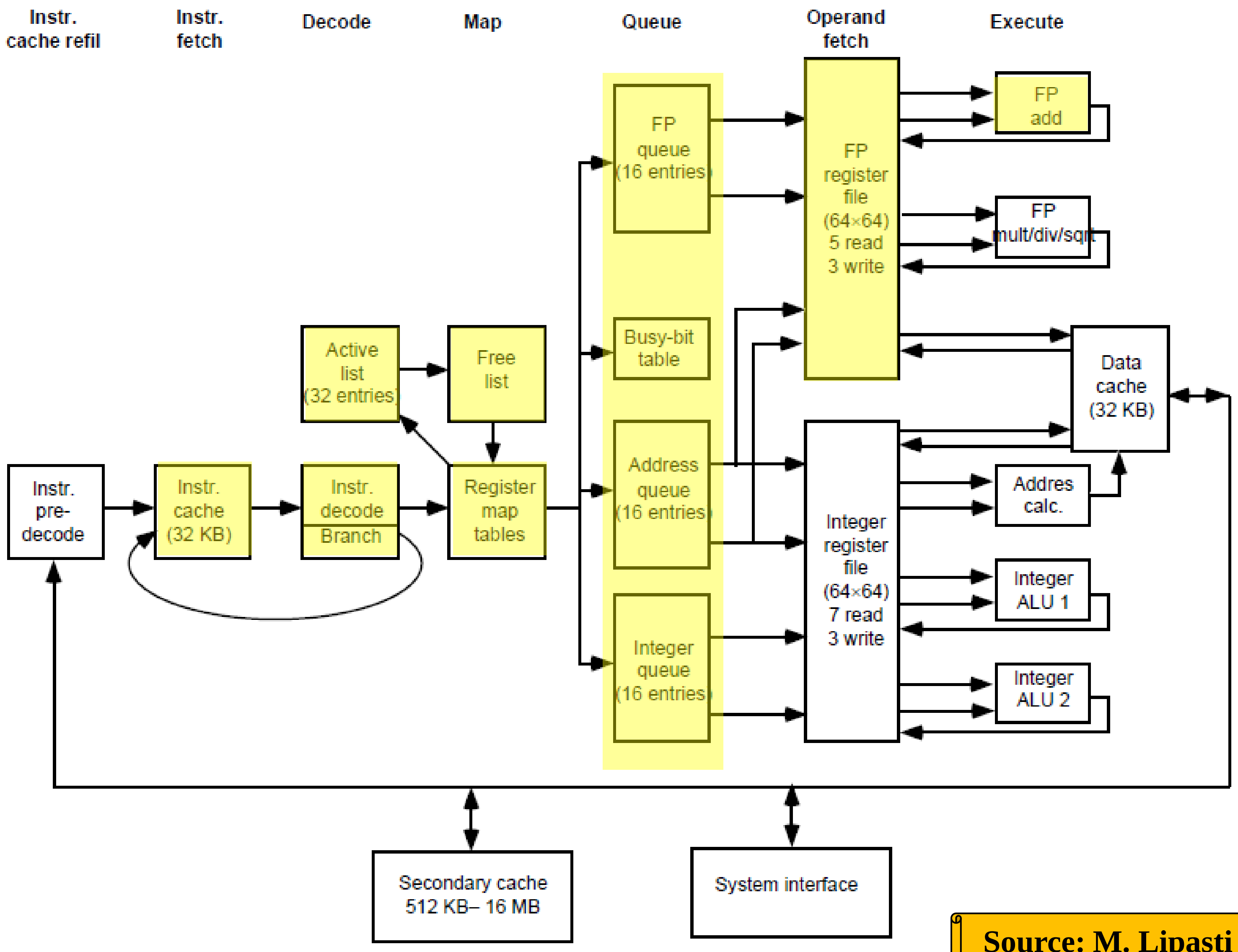


Source: M. Lipasti @ U Wisc.

R-10000 Characteristics

- 298 mm^2 , 0.35 μm, 6.8 M transistors (2.4M CPU + 4.4M Cache)
- Fetches and decodes 4 instrns. every cycle.
- Out-of-order issue/execution.
- 5 Fus
  - Non-block load/store unit
  - 2 Integer Unit
  - 1 FP Adder, and 1 FP mult.
- 32K I$ + 32K D$ (2-way associative).
- External 2-way L2 $ (128-bit sync. Bus)
- Branch prediction: 512 entry 2-bit BHT.
- Speculative execution upto 4 pending branches.

MIPS R10000 Issue Queues

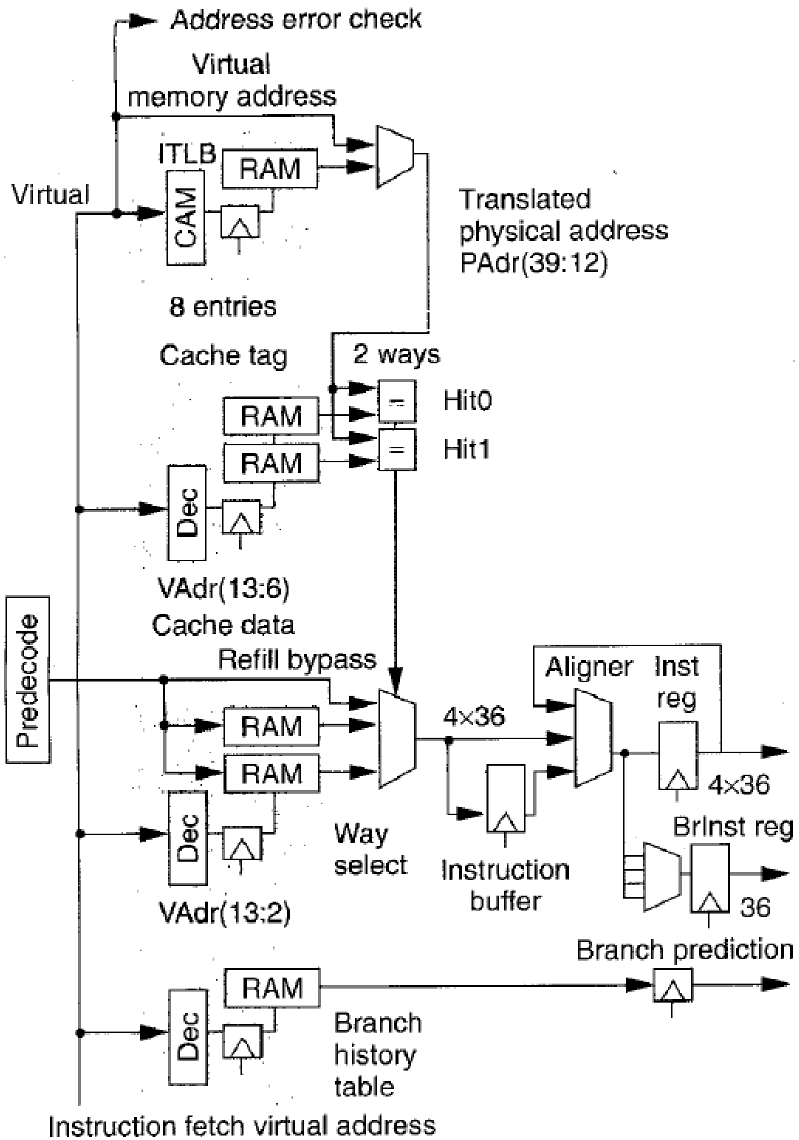


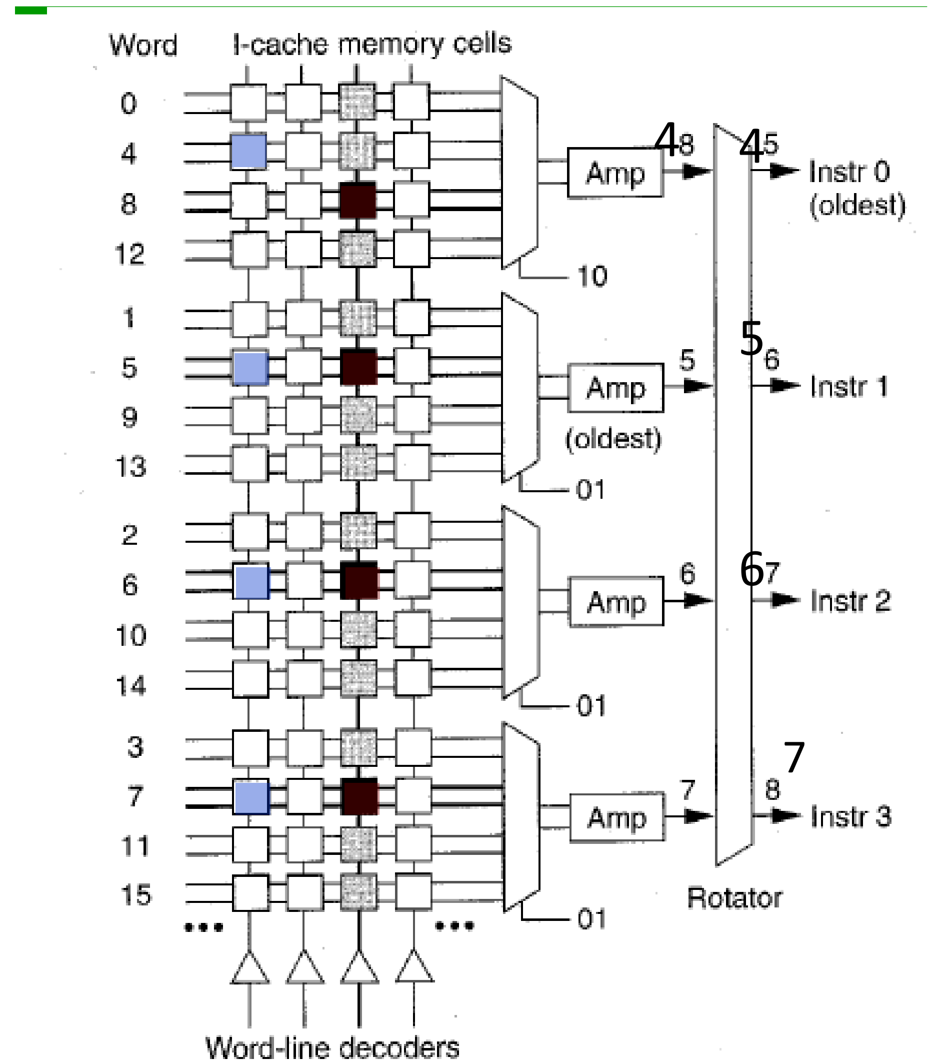
Source: M. Lipasti @ U Wisc.

Instruction Fetch

- Multiple instructions are fetched from I-Cache into Instrn. Buffer.
  - E.G., R10K fetches 4 instrns. (word aligned) within a 16-word I-Cache line.
- Fetch bandwidth might be lower due to
  - Branch instrn. in the fetched instrns.
  - Instrn. fetch starting from the middle of a cache line
  - I-Cache miss

MIPS R10000 – Instruction Fetch

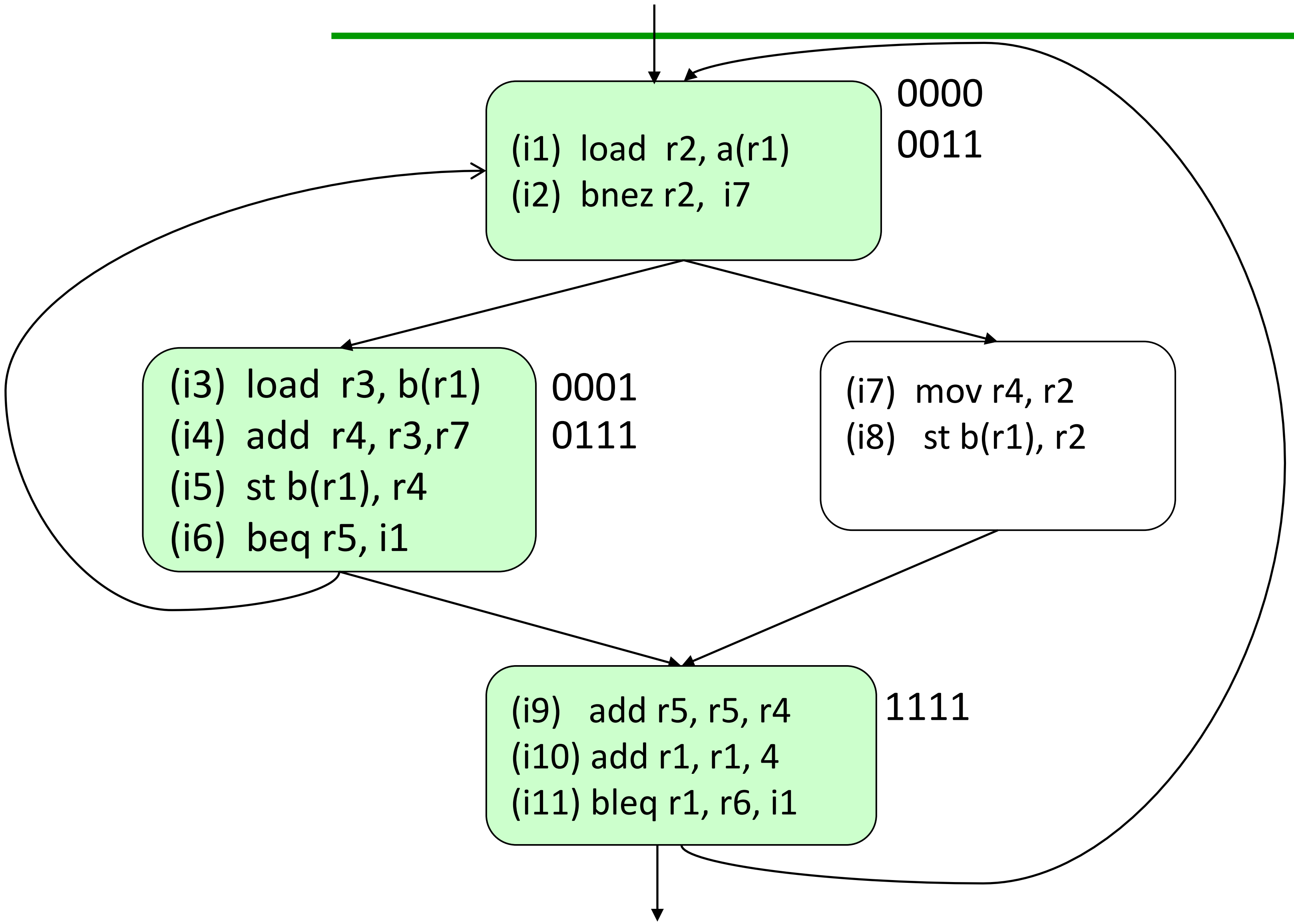




Branch Prediction

- Branch prediction
  - Predecode logic to identify branch instrns. early
  - Branch prediction (through Branch History Table) using PC value (and previous pattern history)
  - R10K has 512-entry 2-bit BHT to support speculative execution upto 4 pending branches.
- Speculatively execute past multiple pending branches.
- Branch stack to save alternate branch addr. and Int. and FP reg. remap tables.
- 4-bit branch mask with each speculatively executed instrn.
- When a branch is detected to be speculated wrong, all instrns. with corpg. branch mask bit set are squashed.

Branch Prediction



Instruction Decode

- Decodes instructions from Instrn. Buffer.
- Detection of RAW hazards and elimination of WAR and WAR hazards through ***Dynamic Register Renaming***

Register Data Dependences

- Program data dependences cause hazards
  - True dependences (RAW)
  - Antidependences (WAR)
  - Output dependences (WAW)

r3 ← r7
r8 ← load (r3)
r3 ← r5 + 4

RAW
WAW
WAR

- RAW dependency must for correctness

- WAR and WAW dependencies – false dependencies – can be eliminated : register renaming

Source: M. Lipasti @ U Wisc.

Instruction Decode

- Decodes instructions from Instrn. Buffer.
- Detection of RAW hazards and elimination of WAR and WAR hazards through ***Dynamic Register Renaming***
  - WAR and WAW hazards occur due to reuse of registers.
  - Limited ***logical*** registers - e.g., 32 Int. and 32 FP Registers.
  - Available ***Physical*** registers/Storage is higher.
  - Replace the logical destination register with a *free* Physical register; any subsequent instrn. which use the value uses the Physical register/storage location.

Register Renaming

- Example:

  mult r2 ← r4, r5
  add r8 ← r2, r4
  sub r4 ← r1, r6
  div r2 ← r19, r20

- Using Physical Registers rename W
  WAW dependences (name depende

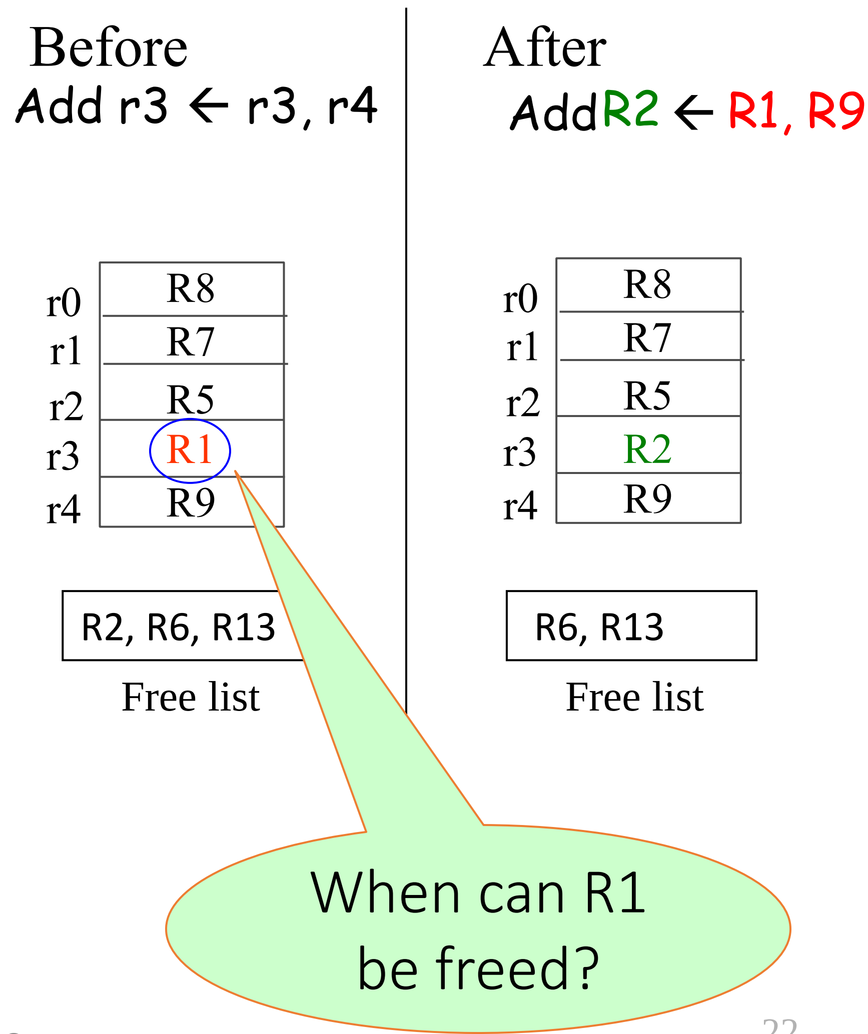
  mult R2 ← R4, R5
  add R8 ← R2, R4
  sub R14 ← R1, R6
  div R25 ← R19, R20

WAR and WAW can be eliminated using Register Renaming.

Register Renaming

- Using Physical Registers (size equals twice the no. of logical registers)
  - A register map table which maintains the association betn. logical and physical registers
  - Free list maintains available free regs.
  - Each source operand reg. is replaced by corpg. Physical reg.
  - Destn. reg. (logical) is assigned a free physical register.
  - Physical reg. is freed after all uses of the value -- when a subseq. instrn. with r3 as destn. commits!

Example: MIPS 10k/12k, DEC-21264



Register Renaming

Before

Add r3 ← r3, r4

r0	R8
r1	R7
r2	R5
r3	R1
r4	R9

R2, R6, R13

Free list

After

Add R2 ← R1, R9

r0	R8
r1	R7
r2	R5
r3	R2
r4	R9

R6, R13

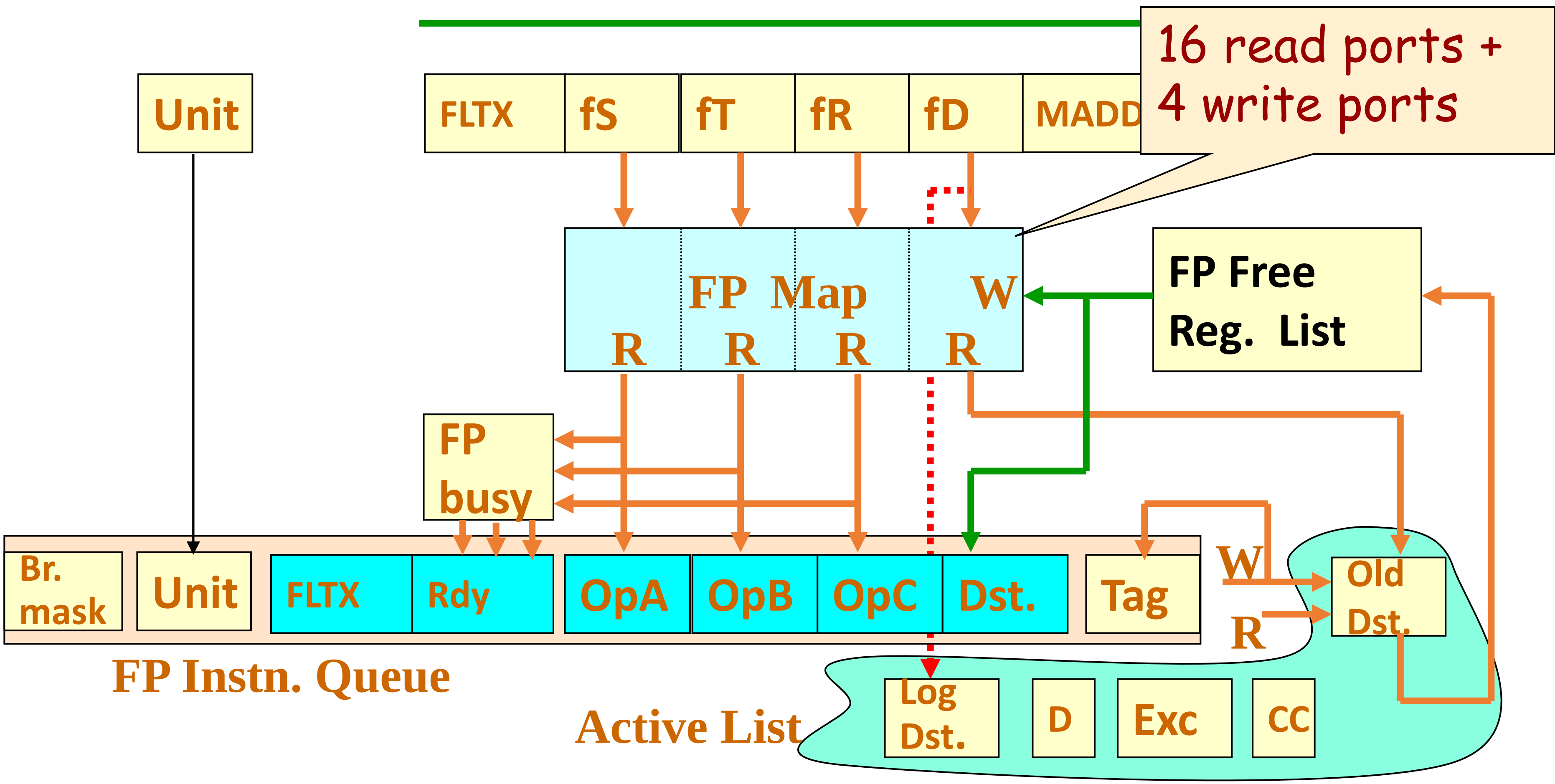
Free list

When can R1 be freed?

```
i1:  add r3 ← r3, r4
i2: sub  r5 ← r3, r1
...

   ##  instrns.with r3
   as source register;
   but not as destn.
   register

...

i10: mult r3 ←  r8, r9
i11: ld r10 ←  M(r3)
```

```
i1:  add R2 ← R1, R9
i2: sub  R6 ← R2, R7
...

...

i10: mult R_ ← R_, R_
i11:  ld R_  ← M(R_ )
```

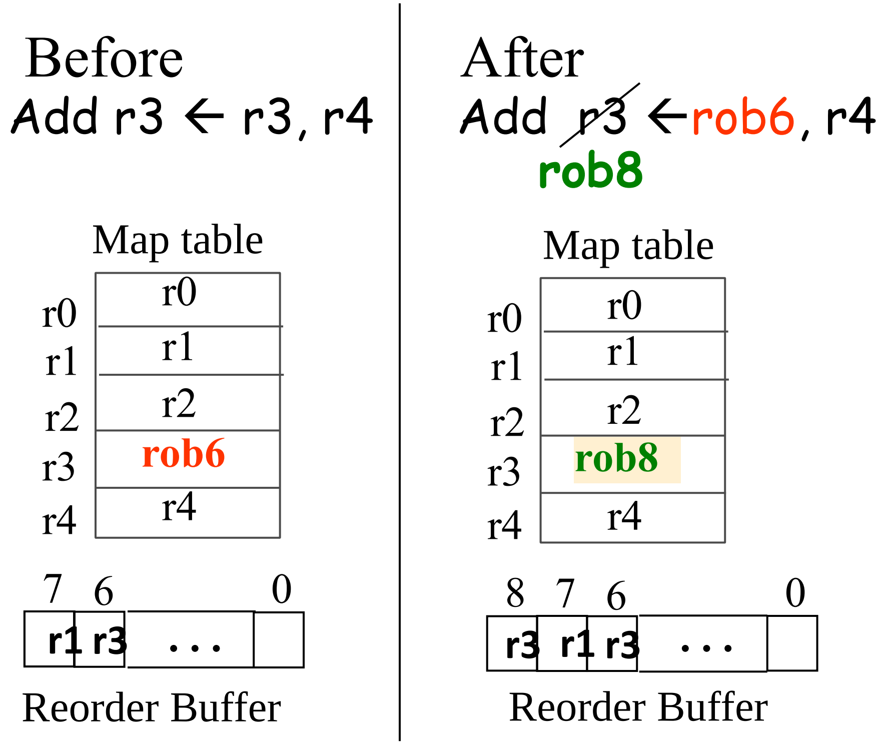
Register Renaming in R10000



- Renaming complexity proportional to IW
- Renaming delay proportional to IW.

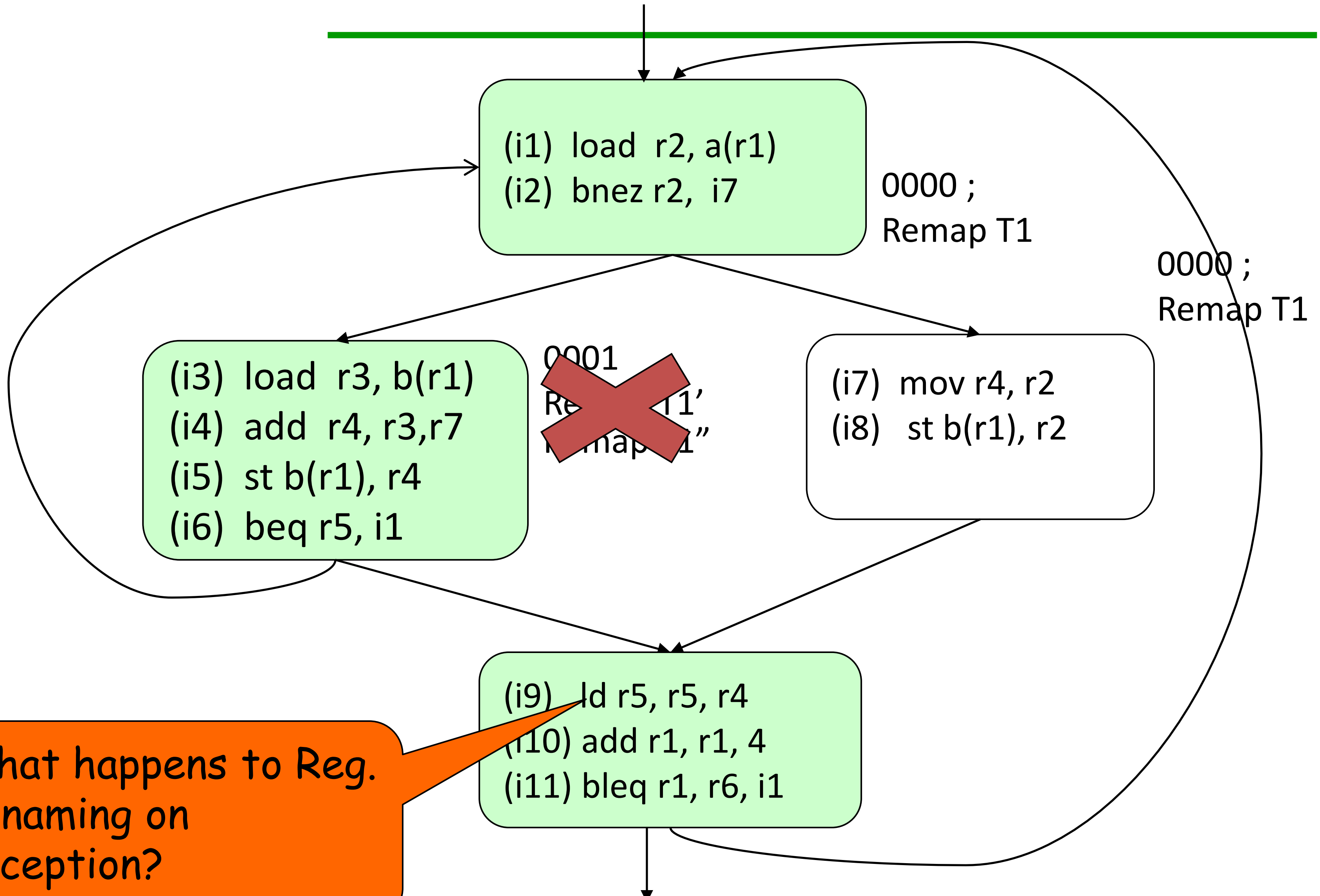
Alternative Register Renaming

- Using *Reorder Buffer* as temporary storage for speculated values.
- No. of Physical Regs. = No. of Logical Regs.
  - Each destn. reg. is assigned entries in the tail of the ROB.
  - Upon instrn. execution, result values are written in ROB.
  - When the instrn. reaches the head of ROB, the value is written in physical (same as logical) reg.
  - Source operands are read from ROB entry or register as indicated by the register map table.



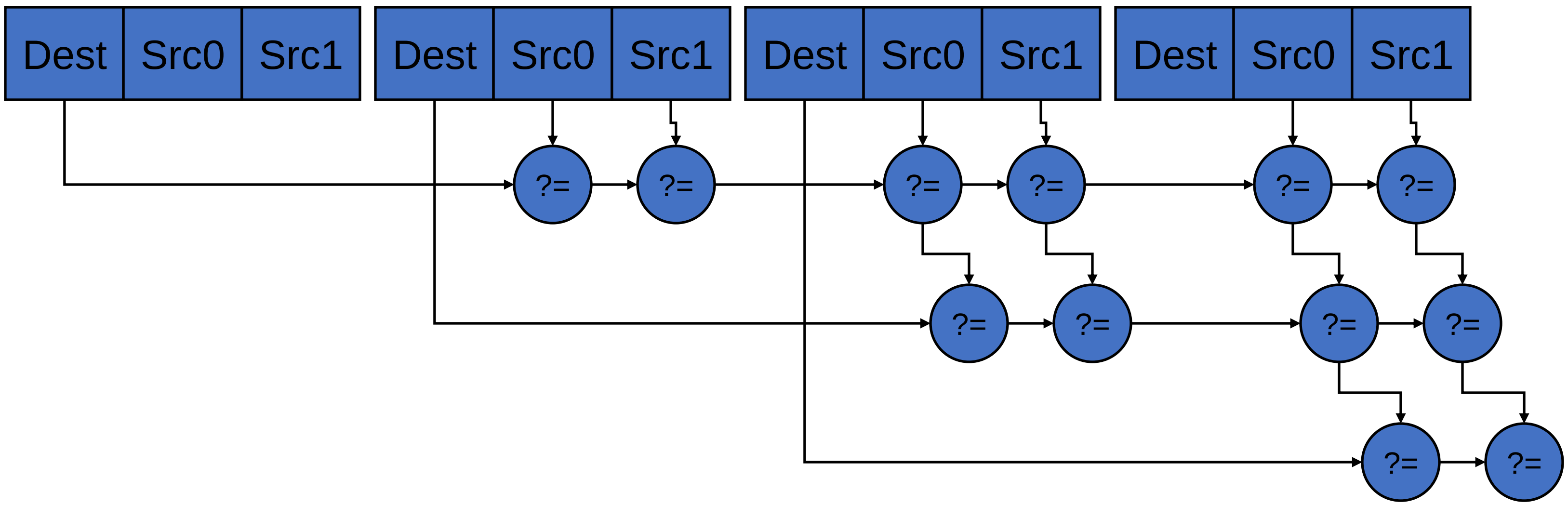
Example: Pentium Pro, HP-PA8000, Power-PC 604, SPARC64

Recovering Renaming on Branch MisPrediction



What happens to Reg. Renaming on exception?

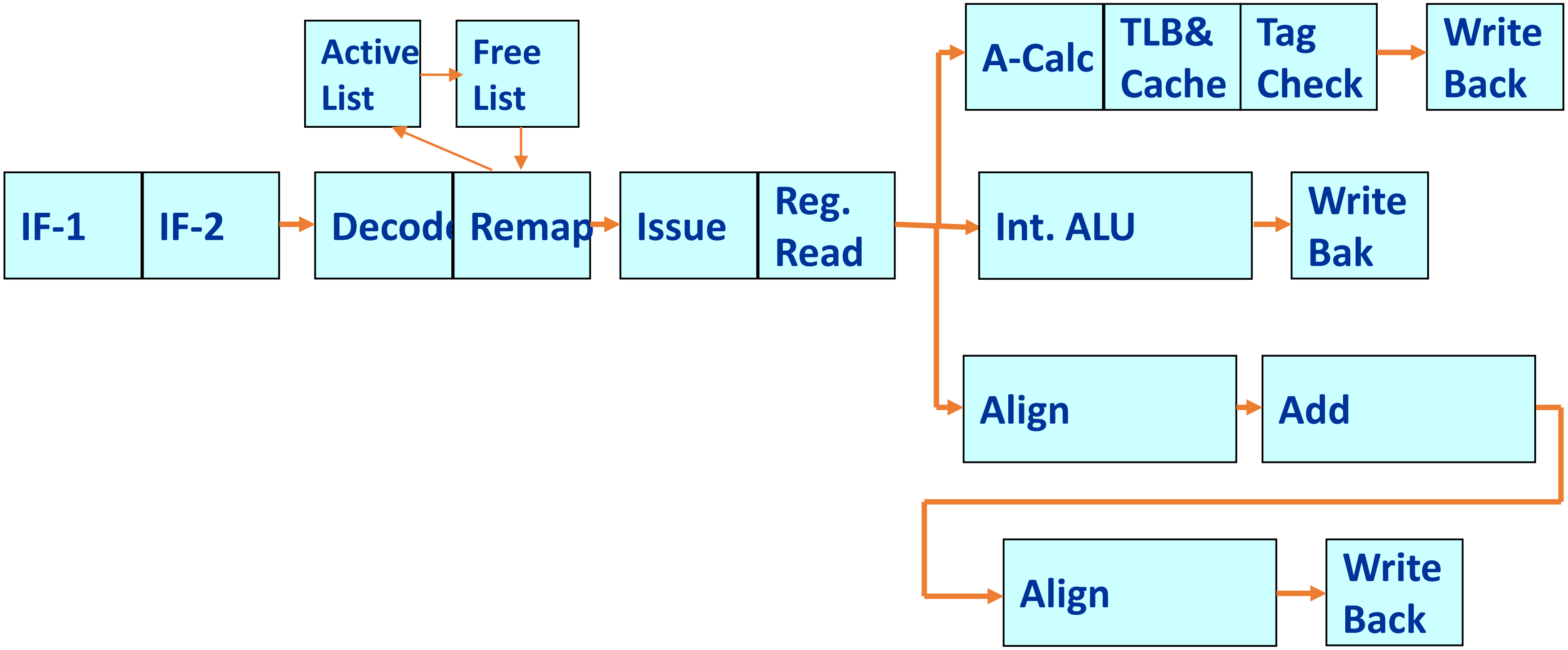
Dependence Checking



- Trailing instructions in fetch group
  - Check for dependence on leading instructions

Source: M. Lipasti @ U Wisc.

Superpipelined Superscalar



In-order vs. Out-of-Order

In-order Issue

load r2 <-- M(r3)

mult r6 <- r4, r2

RAW

sub r8 <- r6, r1

Stalls

store M(r9) <- r8

Stalls

add r3 <- r3, 4

Stalls

add r9 <- r9, 4

Example: MIPS-8000, SPARC, DEC 21064, 21164

Out-of-Order Issue

load r2 <- M(r3)

mult r6 <- r4, r2

sub r8 <- r6, r1

store M(r9) <- r8

Stalls

add r3 <- r3, 4

Stalls

add r9 <- r9, 4

proceeds to FU for exec.

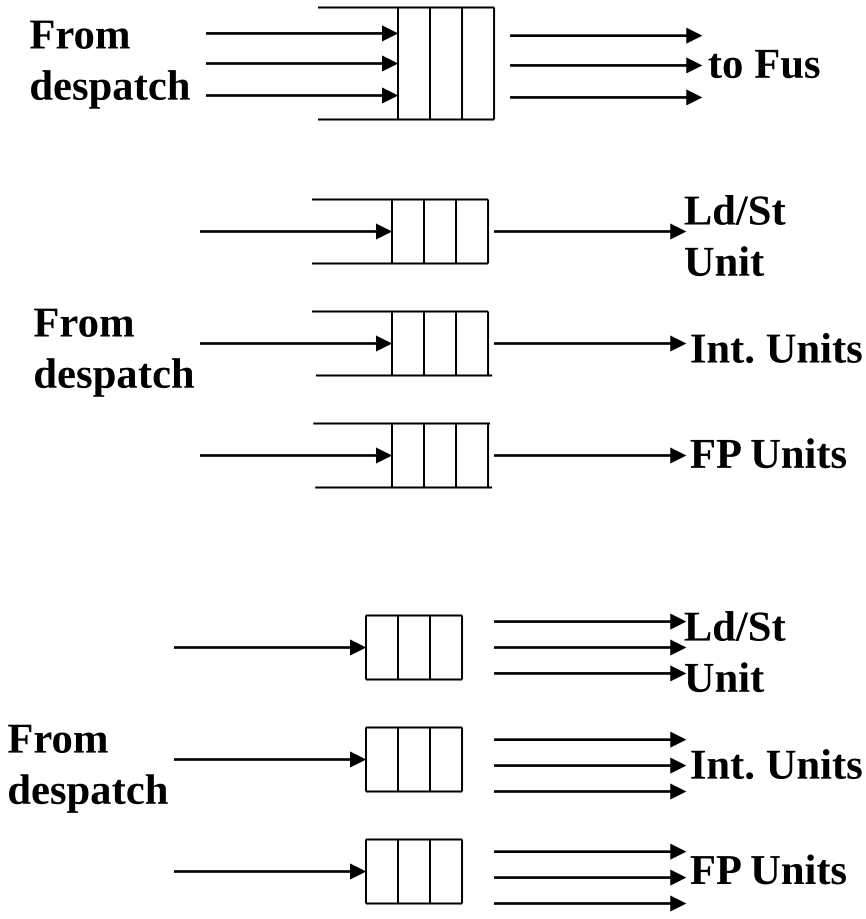
Example: MIPS-10000, PowerPC620, DEC-21264

Instruction Dispatch & Issue

- Decoded and renamed instns. are dispatched to
  - Instruction Queues
    - Single instrn. queue for all instrn. types (typically in *in-order* issue processors!)
    - Different queues for each instrn. type (e.g., Integer, FP, and Load/Store) -- typically in *out-of-order* issue processors!

or

  - Reservation stations for each each FU or FU type (typically for out-of-order issue processors)

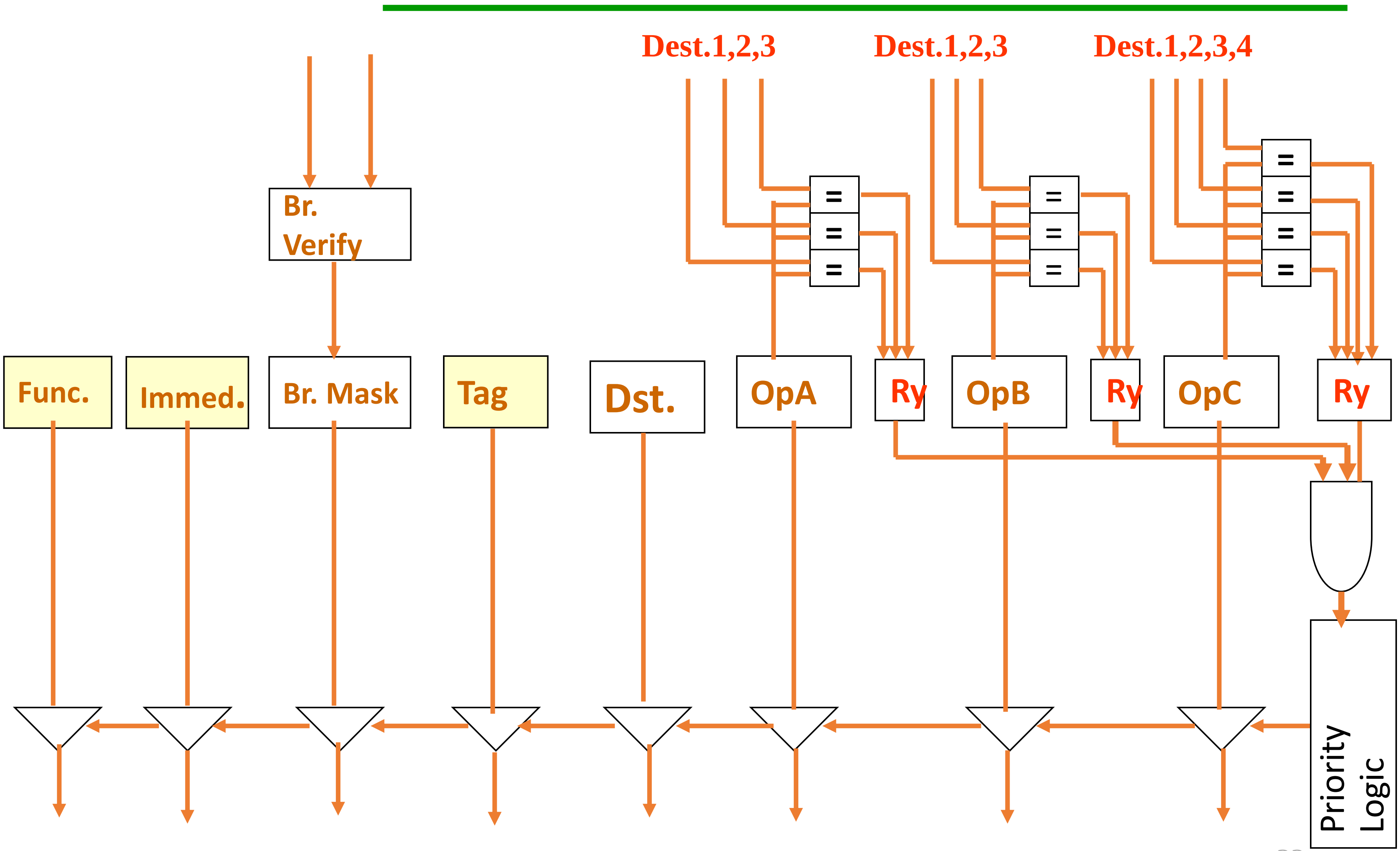


Instruction Issue: Wakeup & Select

- Instrn. Wakeup: waits in instrn. queue/ reservation station for
  - data dependences to be satisfied (check for ready bits in the Instrn. Queue/Res. Station entry).
  - resource (Functional Unit) to be available.

- Instruction Select:
  - Among the ready instrns., a subset is selected (based on ***priority***) to be issued to the FUs.

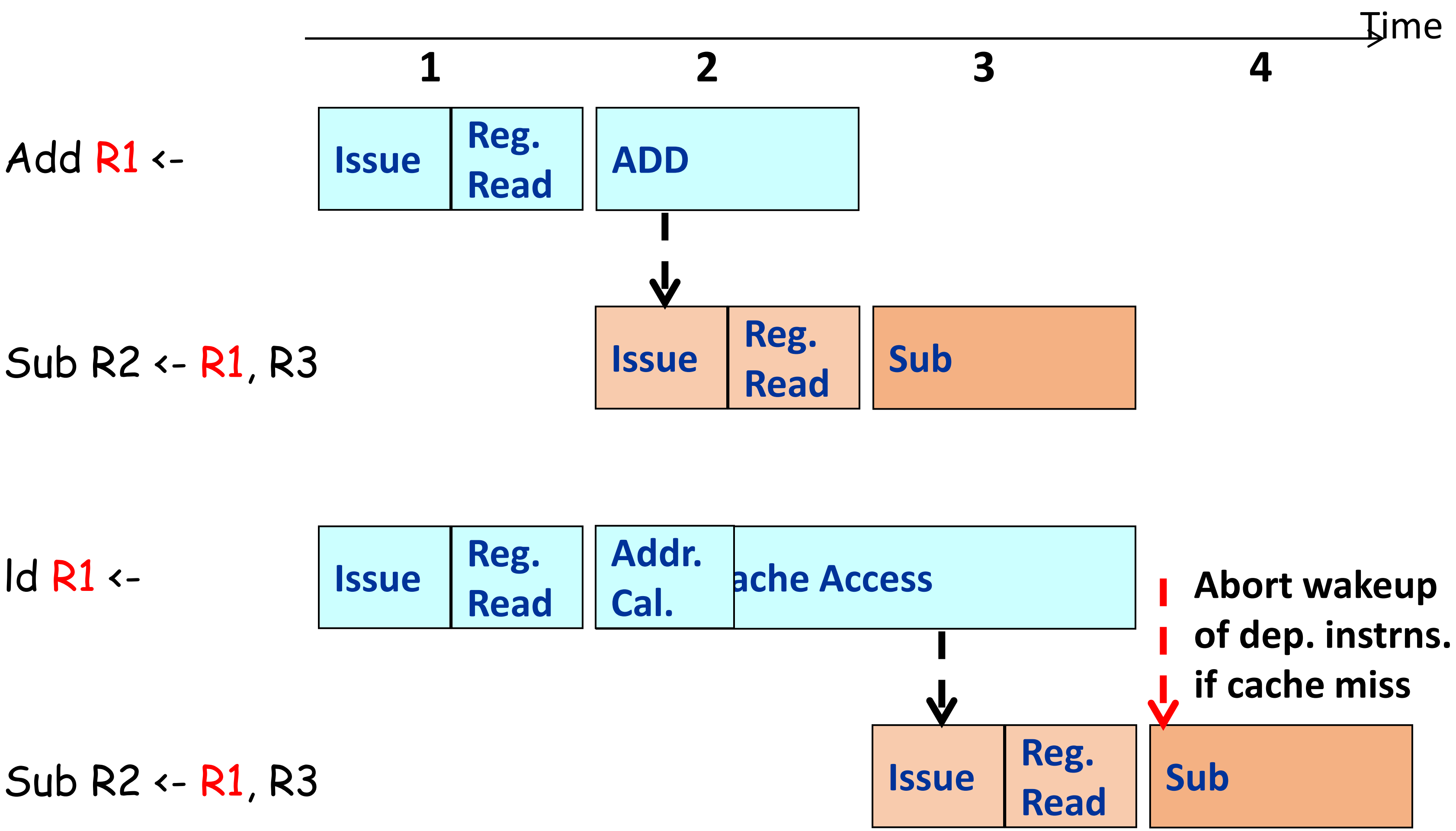
Wakeup & Select in R10000



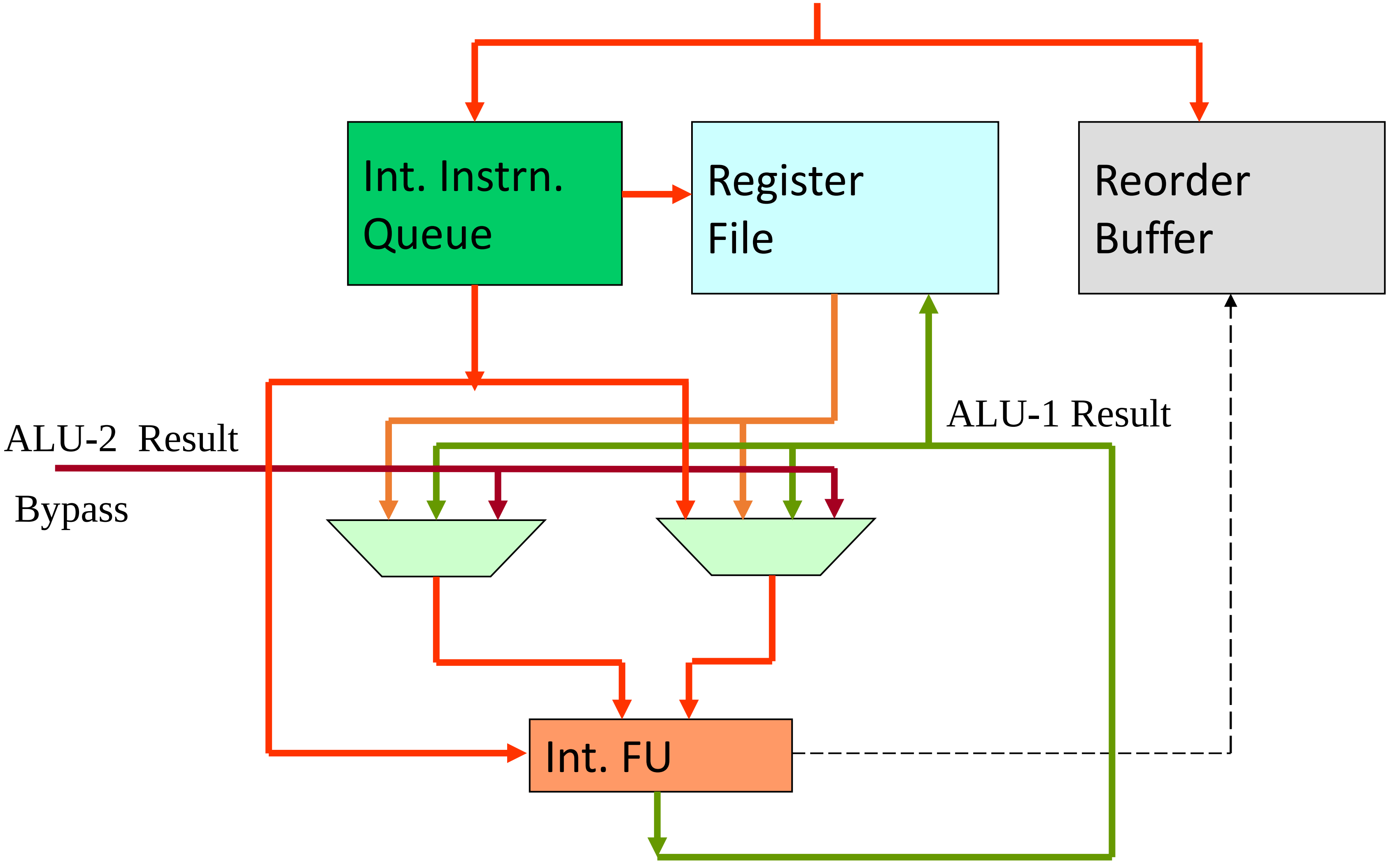
Wakeup & Select

- Wakeup phase of the issue stage :
  - The Result tag from the completing instructions is broadcast to all waiting instructions.
  - All instructions compare all tags and set ready bits appropriately.
  - Typically implemented as a CAM
  - Wakeup logic complexity proportional to Instrn. Window size and Issue width.
- Select phase of the issue stage :
  - Choose all or some of the ready instructions for execution depending on age and EU availability.
  - Selection logic complexity proportional to Window size
- Challenges
  - Wakeup done ***tentatively*** when producer instrn. is selected
  - Wakeup and select phases are complex; Cannot be pipelined if dependent instrns. are to be issued in successive cycles.

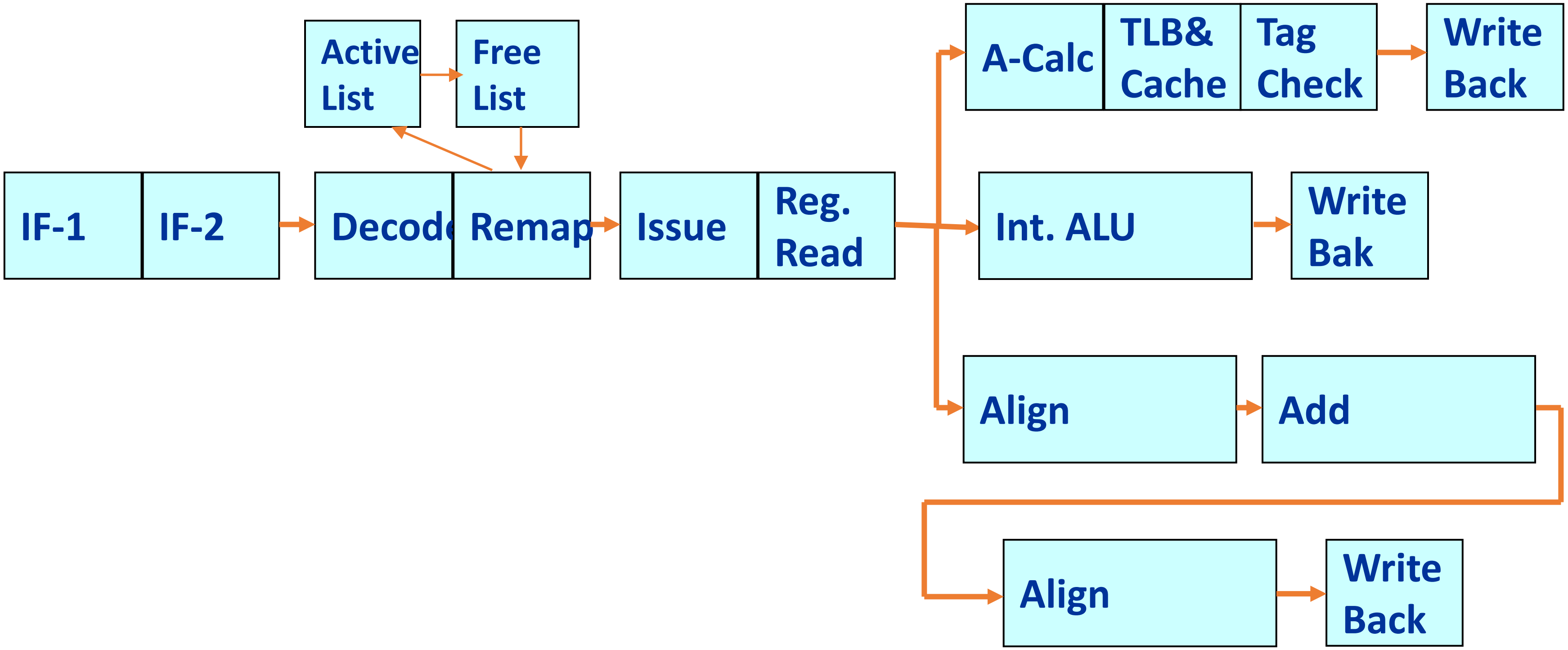
Issue of Dependent Instructions



Instn. Issue & Execution in R10000



Superpipelined Superscalar



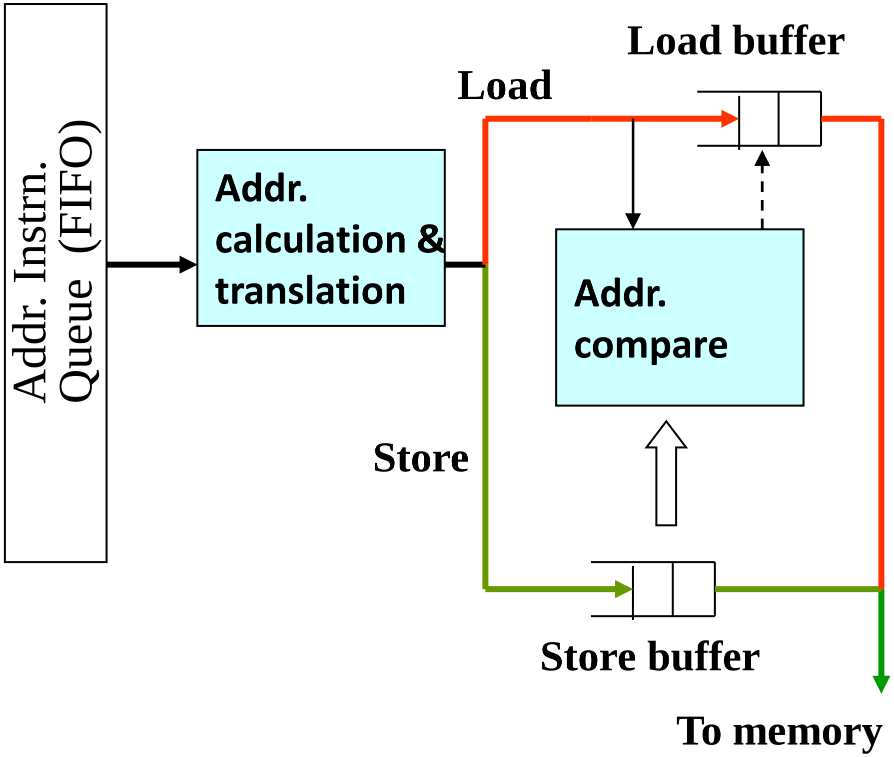
Load/Store Queue

- Memory ops. involve address *calculation* and *translation*
  - use of TLB for translation.
- Load/Store queue is typically FIFO to ensure load/stores dependences

  Store M(R8) ← R3

  load R6 ← M (R16)

Dependent on store or not?



- Memory renaming (unlike reg. renaming) is difficult.
- Store instrns. wait in Store buffer until they become head of Active List.

Commit Stage

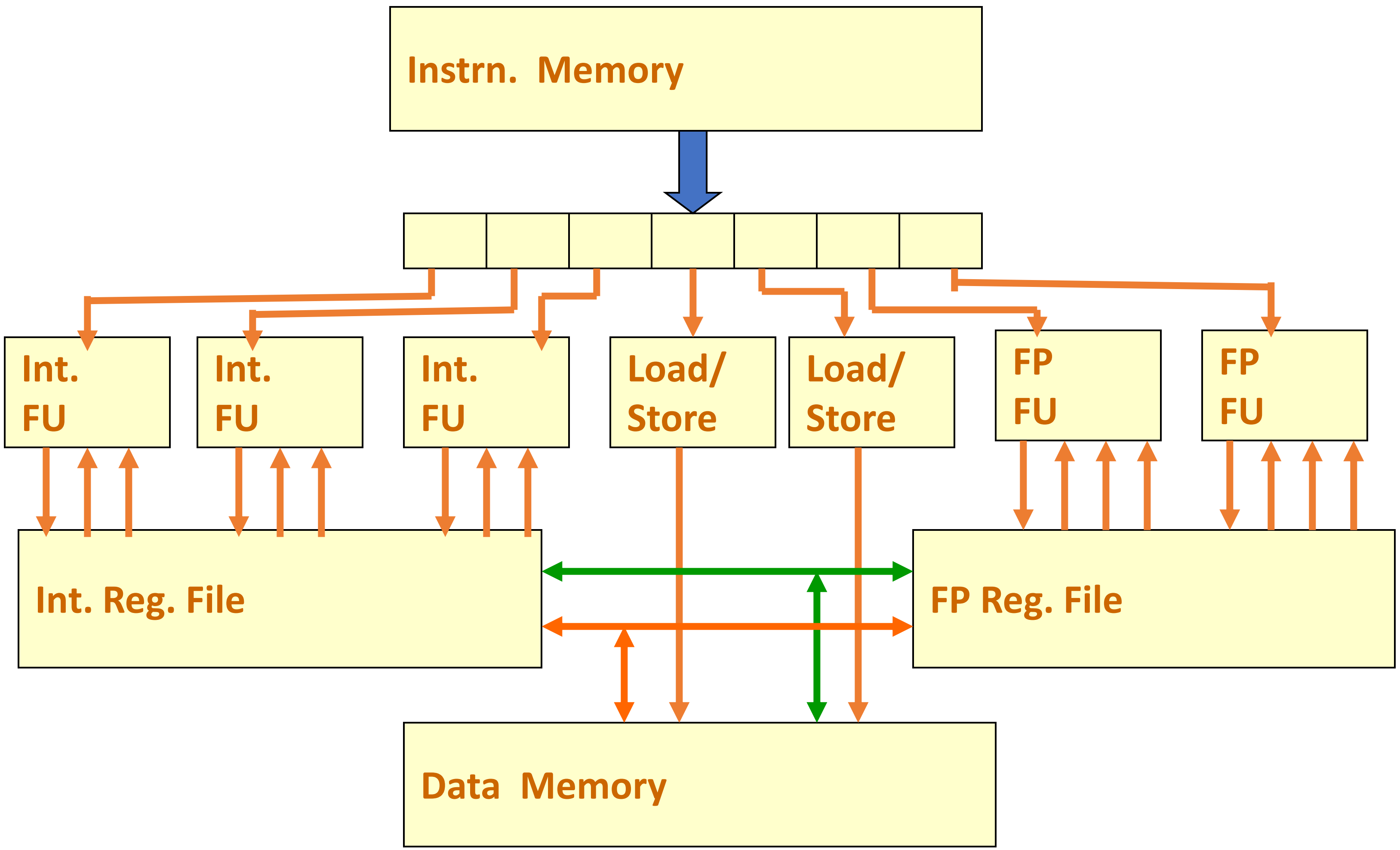
- Typically, instructions are committed in program order (from Reorder Buffer)
  - To maintain appearance of sequential exec. in ***speculative*** and ***out-of-order*** execution.
  - To maintain ***precise interrupts.***
- For branch misprediction, speculatively executed instrns. and their effects are annulled by
  - reloading the register map table (saved on branch prediction) .
  - In physical reg. renaming, restoring reg. map table suffices.
  - squashing a part of the Reorder Buffer (if ROB is used for register renaming)

VLIW Architecture: Motivation

- Fast, simple decoding and instrn. issue logic that can issue multiple operations in a cycle.
- Compiler to pack to multiple ops. in a single instrn. (inspired by ***horizontal micro-programming***)
- No hazard detection (interlocking) -- compiler to ensure correct semantics.
- Reduce the hardware needed for dynamic instrn. scheduling

Smart compiler and a Dump (but fast) processor!

VLIW Processor Organization



Cydra 5

- Heterogeneous multiprocessor consisting of a Generic Proc. (multiple procs.) and a Numeric Processor (VLIW).
  - 256-bit 7-wide instrn. (1 FAdd, 1FMult, 2 Mem, 2 Int. (Addr. Arith.) and 1 Branch instrn.
  - Instrn. Format supports Multi-op and UniOp mix.
  - Multicycle NoOP instrn.
  - Predicated Execution: Result and exception suppressed if predicate is false
  - Turns control dependence into data dependence.

An Example Program

- Consider

```
for (i=0; i < 100; i++)
    a[i] = a[i] + s;
```

- Assembly code

```
L:LD  F0, 0(R1)          ; 1 stall cycle
  ADDD  F4,F2,F0         ; 2 stall cycle
  ST   0(R1), F4
  ADD  R1, R1, #8
  Sub R2,R2, #1
  Bnez R2, L             ; 1 branch stall cycle
```

- VLIW with 1 Mem., 1 Int., 1FP and 1 Branch per instrn. For 5-times unrolled loop.

An Example VLIW Program

Mem. Op	*FP Op.*	*Int. Op.*	*Branch*
LD F0, 0(R1)	**--**	**Sub R2, R2 #1**	**--**
--	**--**	**--**	**--**
--	**ADDD F4,F2, F0**	**Add R1, R1, #8**	**--**
--	**--**	**--**	**--**
--	**--**	**--**	**Bnez R2, L**
ST -8 (R1), F4	**--**	**--**	**--**

1 cycle

2 cycle

- 6 Cycles even on the VLIW archiitecture!
- Unroll the loop a few times (5 times) and schedule?

An Example VLIW Program

Mem. Op	*FP Op.*	*Int. Op.*	*Branch*
LD F0, 0(R1)	--	--	--
LD F6, 8(R1)	--	--	--
LD F10, 16(R1)	ADDD F4,F2, F0	--	--
LD F14, 24(R1)	ADDD F8,F2, F6	--	--
LD F18, 32(R1)	ADDD F12,F2,F10	--	--
ST 0(R1), F4	ADDD F16,F2,F14	--	--
ST 8(R1), F8	ADDD F20,F2,F18	Sub R2,R2, #5	--
ST 16(R1), F1	--	Add R1,R1,#40	--
ST –16(R1),F16	--	--	Bnez R2, L
ST –8(R1), F20	--	--	--

1 cycle

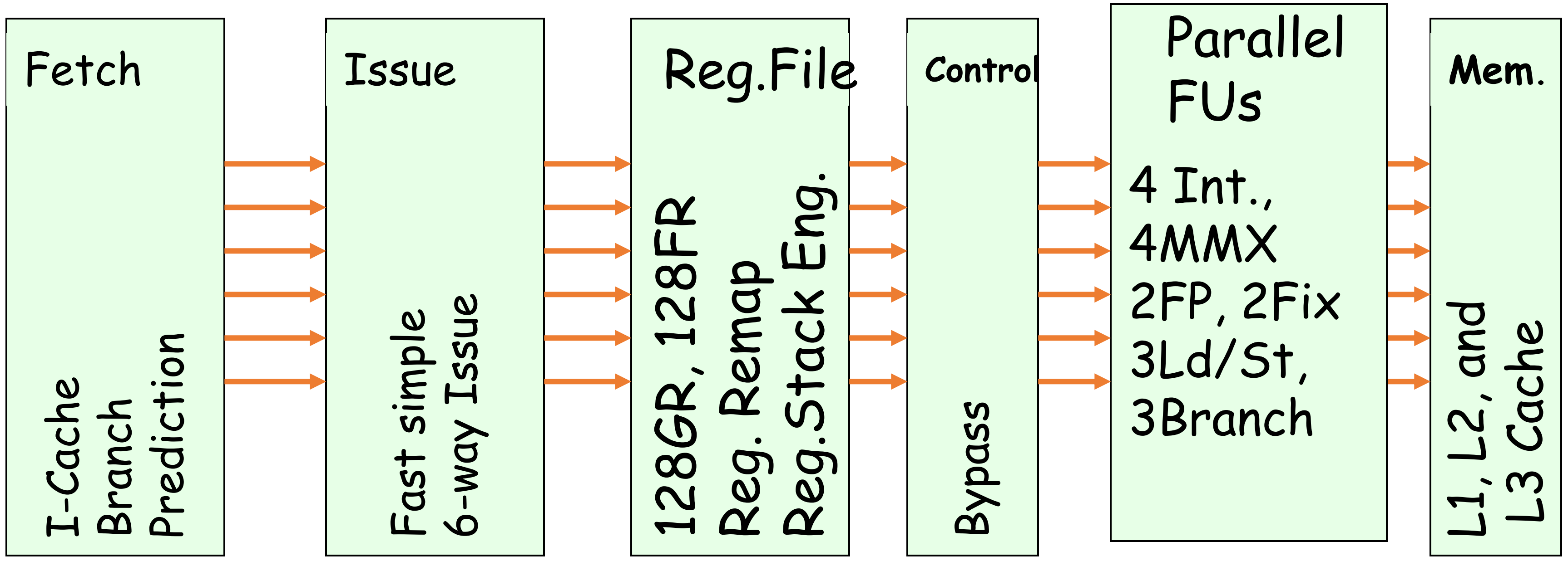
2 cycle

Avg. no. of operations/instrn. = 18/10 = 1.8

EPIC - IA64 Architecture

- Explicitly Parallel Instruction Computing:
  - Compiler packs independent instrns. in a 128-bit bundle consisting three 41-bit instrn. and a 5-bit template (for easy decoding).
  - Stop or No-Stop at the end of the bundle.
  - Instrn. independence explicitly conveyed.
  - Multiple bundles can be issued in a cycle.
  - Hint bits (for cache hit, branch behavior, …)
  - Guarded execution, e.g., if (p5) r1 = r2 + r3
  - 128 Int. and 128 FP registers.
  - Register stack (for procedure call) and rotating registers (for software pipelining)

Itanium Microarchitecture



Predicted Execution

High-Level Code

```
if r1 != 0
  r3 = r2 + r3
else
  r4 = r2 – r3
```

Assembly Code

```
i1: beqz r1, i4
i2: r3 = r2 + r3
i3: br i5
i4: r4 = r2 – r3
i5:
```

Predicated Code

```
i1: seqz r1, p1
i2: (!p1) r3 = r2 + r3
i4: (p1) r4 = r2 – r3
```