18-640 Foundations of Computer Architecture

Lecture 1:
"Introduction To Computer Architecture"

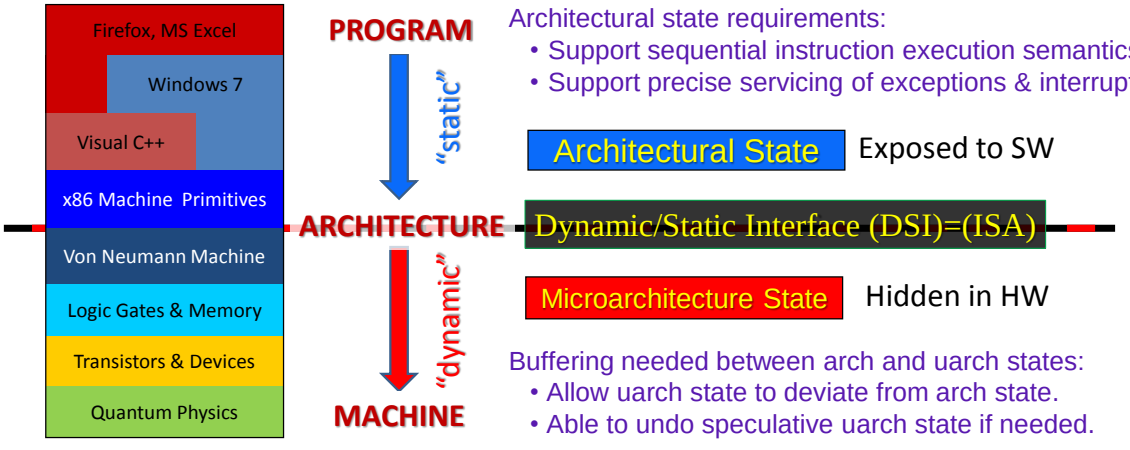
A. Instruction Set Architecture (ISA)
- a. Hardware / Software Interface
- b. Dynamic / Static Interface (DSI)

B. Historical Perspective on Computing
- a. Major Epochs
- b. Processor Performance Iron Law (#1)
- c. Course Coverage

C. "Economics" of Computer Architecture
- a. Amdahl's Law and Gustafson's Law
- b. Moore's Law and Bell's Law

Electrical & Computer ENGINEERING

Computer Architecture: Dynamic-Static Interface



DSI = ISA = a contract between the program and the machine.

"Iron Law" of Processor Performance

$$\text{1/Processor Performance} = \frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

(inst. count) (CPI) (cycle time)

Architecture → **Implementation** → **Realization**

Compiler Designer | Processor Designer | Chip Designer

- In the 1980's (decade of pipelining):
  - CPI: 5.0 → 1.15
- In the 1990's (decade of superscalar):
  - CPI: 1.15 → 0.5 (best case)
- In the 2000's:
  - we learn the power lesson

Four Foundational "Laws" of Computer Architecture

- **Application Demand** (PROGRAM)
  - **Amdahl's Law** (1967)
    - Speedup through parallelism is limited by the sequential bottleneck
  - **Gustafson's Law** (1988)
    - With unlimited data set size, parallelism speedup can be unlimited
- **Technology Supply** (MACHINE)
  - **Moore's Law** (1965)
    - (Transistors/Die) increases by 2x every 18 months
  - **Bell's Law** (1971)
    - (Cost/Computer) decreases by 2x every 36 months

18-640 Foundations of Computer Architecture

Lecture 2:
"Review of Pipelined Processor Design"

A. Pipelining Fundamentals
B. Pipelined Processor Organization
   a. Balancing Pipe Stages
   b. Unifying Instruction Types
   c. Resolving Pipeline Hazards
C. Pipelined Processor Performance
   a. ALU Instruction Interlock & Penalty
   b. Load Instruction Interlock & Penalty
   c. Branch Instruction Interlock & Penalty

Electrical & Computer ENGINEERING

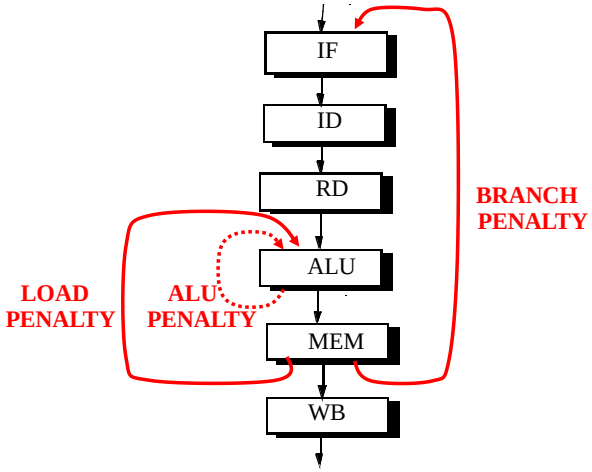
Instruction Pipeline Design

- Uniform Sub-computations ... NOT!
  ⇒ balancing pipeline stages
  - stage quantization to yield balanced pipe stages
  - minimize internal fragmentation (some waiting stages)
- Identical Computations ... NOT!
  ⇒ unifying instruction types
  - coalescing instruction types into one multi-function pipe
  - minimize external fragmentation (some idling stages)
- Independent Computations ... NOT!
  ⇒ resolving pipeline hazards
  - inter-instruction dependence detection and resolution
  - minimize performance lose due to pipeline stalls

Inter-Instruction Hazards

Pipe Stage	ALU Inst.	Load inst.	Store inst.	Branch inst.
1. IF	I-cache PC<PC+4	I-cache PC<PC+4	I-cache PC<PC+4	I-cache PC<PC+4
2. ID	decode	decode	decode	decode
3. RD	read reg.	read reg.	read reg.	read reg.
4. ALU	ALU op.	addr. gen.	addr. gen.	addr. gen. cond. gen.
5. MEM	-----	read mem.	write mem.	PC<-br. addr.
6. WB	write reg.	write reg.	-----	-----

3 Major Penalty Loops of (Scalar) Pipelining



Performance Objective: Reduce CPI as close to 1 as possible.

18-640 Foundations of Computer Architecture

Lecture 3:
"From Pipelined to Superscalar Processors"

A. Limitations of Scalar Pipelined Processors
   - IBM RISC Experience

B. Modern Superscalar Processor Organization
   - Parallel, Diversified, and Dynamic Pipelines

C. Limits on Instruction Level Parallelism (ILP)
   - Motivation for Superscalar Processors

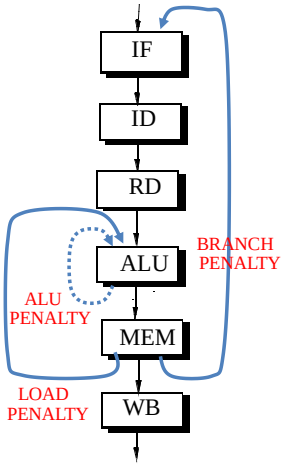
D. Computer Architecture & Operating Systems

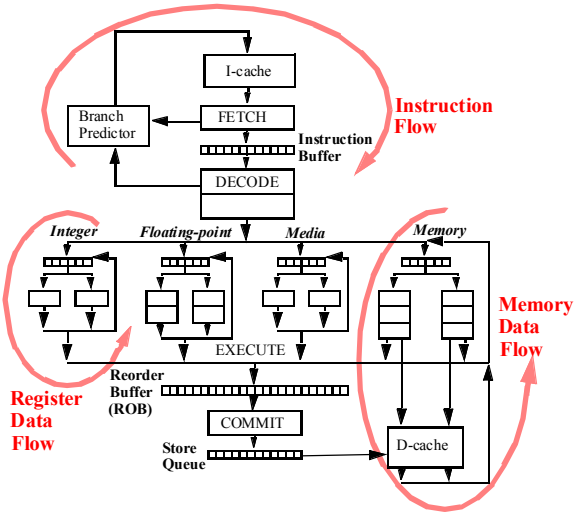
Electrical & Computer ENGINEERING

Limitations of Scalar Pipelined Processors

- Upper Bound on Scalar Pipeline Throughput
  Limited by IPC = 1
  ➔ Parallel Pipelines
- Inefficient Unification Into Single Pipeline
  Long latency for each instruction
  Hazards and associated stalls
  ➔ Diversified Pipelines
- Performance Lost Due to In-order Pipeline
  Unnecessary stalls
  ➔ Dynamic Pipelines

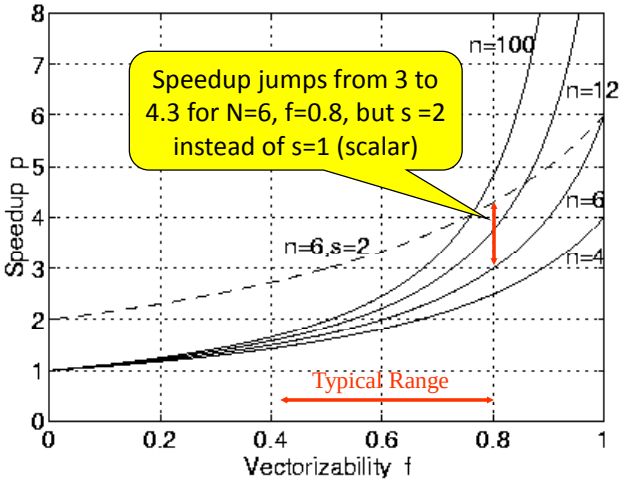
Impediments to Superscalar Performance





[Tilak Agerwala and John Cocke, 1987]

Motivation for Superscalar Design



Limits on Instruction Level Parallelism (ILP)

Weiss and Smith [1984]	1.58
Sohi and Vajapeyam [1987]	1.81
Tjaden and Flynn [1970]	1.86 (Flynn's bottleneck)
Tjaden and Flynn [1973]	1.96
Uht [1986]	2.00
Smith et al. [1989]	2.00
Jouppi and Wall [1988]	2.40
Johnson [1991]	2.50
Acosta et al. [1986]	2.79
Wedig [1982]	3.00
Butler et al. [1991]	5.8
Melvin and Patt [1991]	6
Wall [1991]	7 (Jouppi disagreed)
Kuck et al. [1972]	8
Riseman and Foster [1972]	51 (no control dependences)
Nicolau and Fisher [1984]	90 (Fisher's optimism)

18-640 Foundations of Computer Architecture

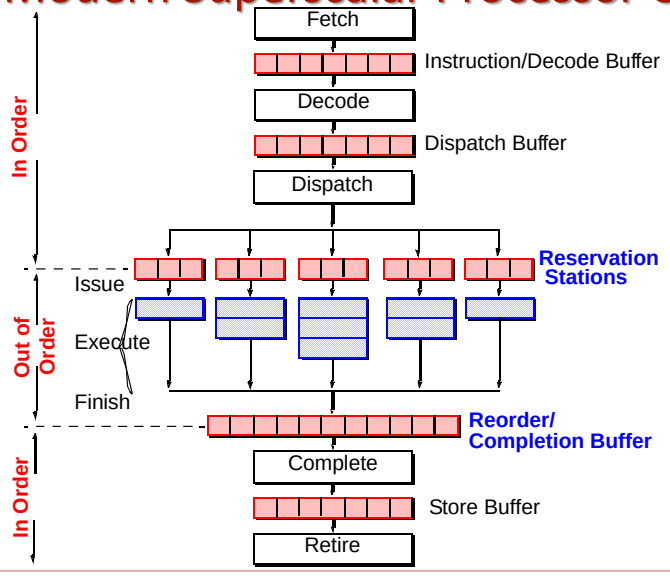
Lecture 4: "Superscalar Implementation and Instruction Flow"

A. Superscalar Pipeline Implementation
   a. Instruction Fetch and Decode
   b. Instruction Dispatch and Issue
   c. Instruction Execute
   d. Instruction Complete and Retire

B. Instruction Flow Techniques
   a. Control Flow Graph
   b. Introduction to Branch Prediction

Electrical & Computer ENGINEERING

A Modern Superscalar Processor Organization



- Buffers provide decoupling
- In OOO designs they also facilitate (re-)ordering
- More details on specific buffers to follow

18-640 Foundations of Computer Architecture

Lecture 5: "Branch Prediction Techniques"

A. Branch Prediction Algorithms
   a. Branch Target Speculation
   b. Branch Direction Speculation
B. Advanced Branch Prediction
C. Wide Instruction Fetch
D. Still More on Branch Predictors

Electrical & Computer ENGINEERING

Dynamic Branch Prediction Tasks

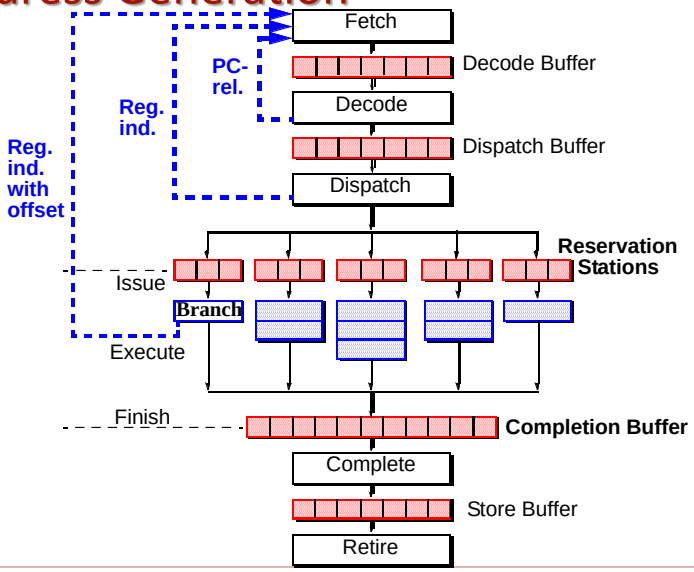
- Target Address Generation
  - Access register
    - PC, GP register, Link register
  - Perform calculation
    - +/- offset, auto incrementing/decrementing

  ⇒ Target Speculation

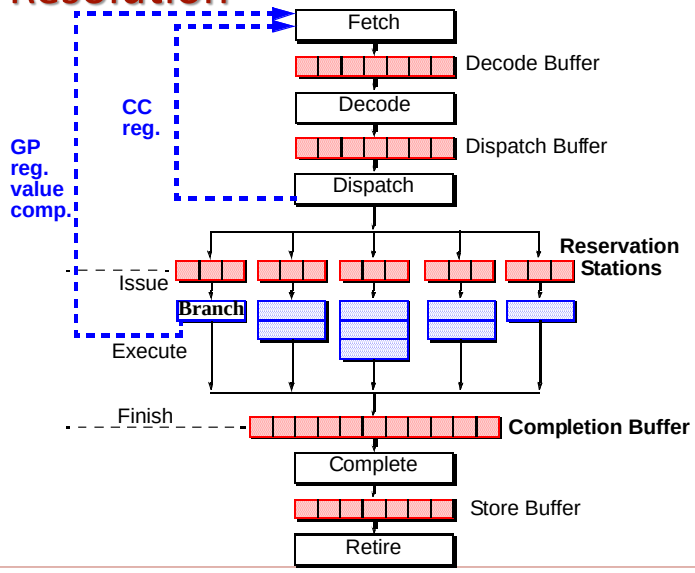
- Condition Resolution
  - Access register
    - Condition code register, data register, count register
  - Perform calculation
    - Comparison of data register(s)

  ⇒ Condition Speculation

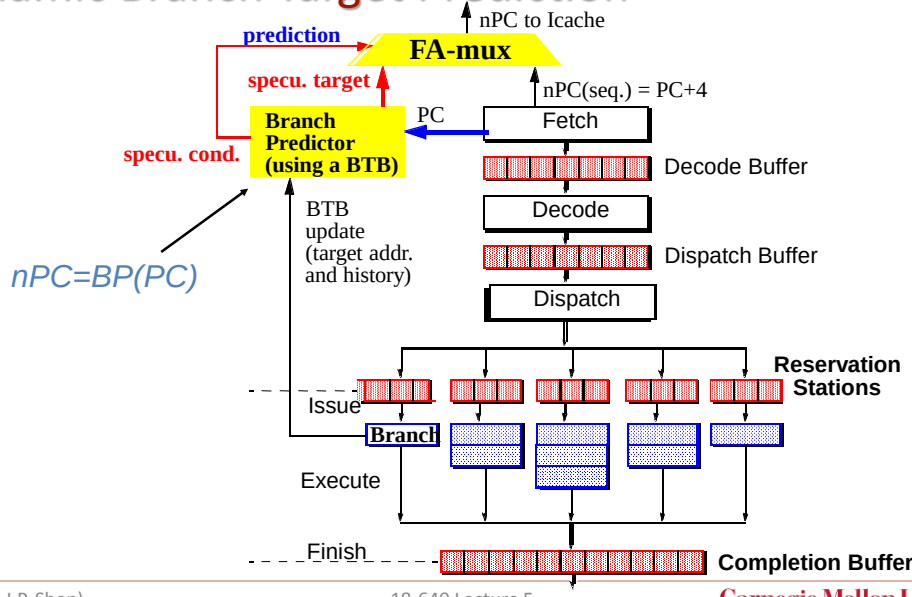
Target Address Generation



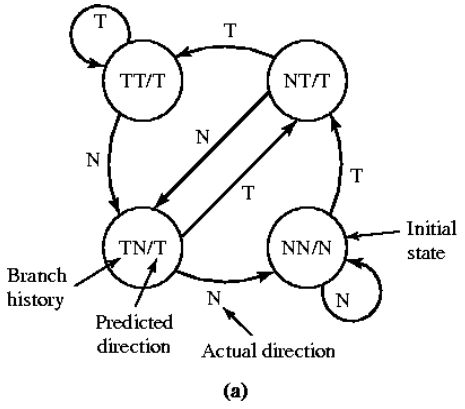
Condition Resolution

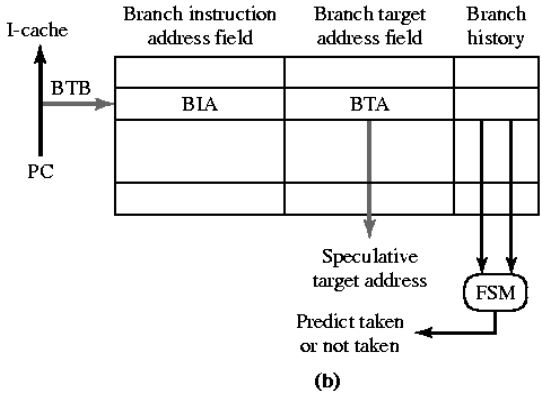


Dynamic Branch Target Prediction

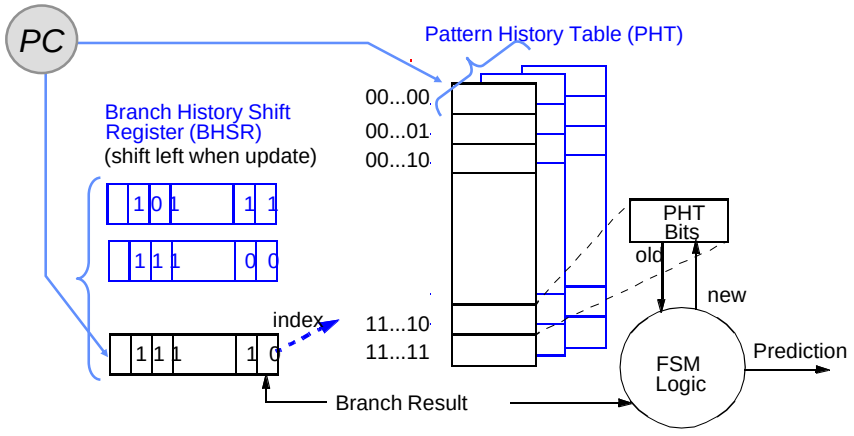


History-Based Branch Prediction

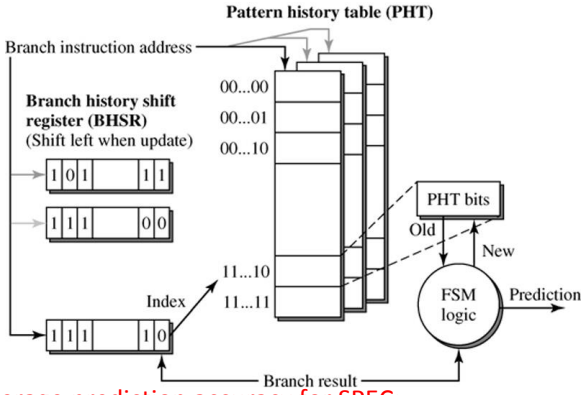




Two-Level Adaptive Branch Prediction



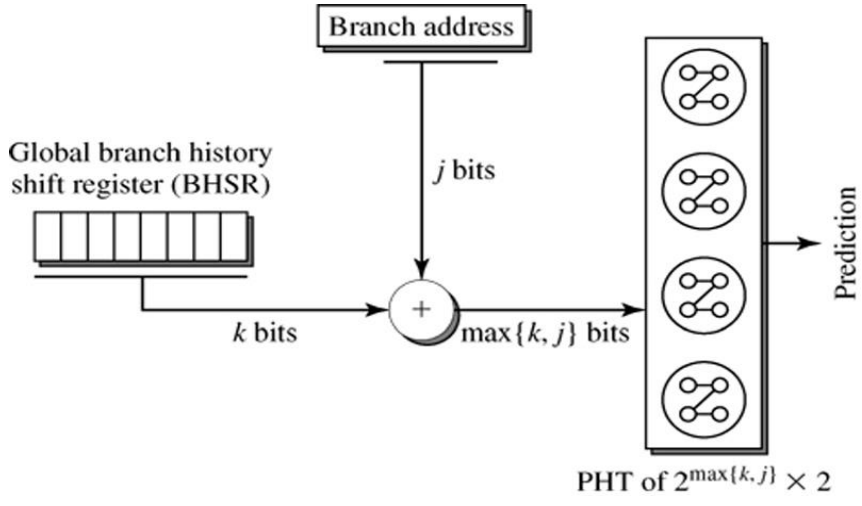
2-Level Adaptive Branch Prediction [Yeh & Patt]



To achieve 97% average prediction accuracy for SPEC:

G (1) BHR: 18 bits;	g (1) PHT: 2^{18} x 2 bits	*total = 524 kbits*
P (512x4) BHR: 12 bits;	g (1) PHT: 2^{12} x 2 bits	*total = 33 kbits*
P (512x4) BHR: 6 bits;	s (512) PHT: 2^{6} x 2 bits	*total = 78 kbits*

Gshare Branch Predictor [Scott McFarling]



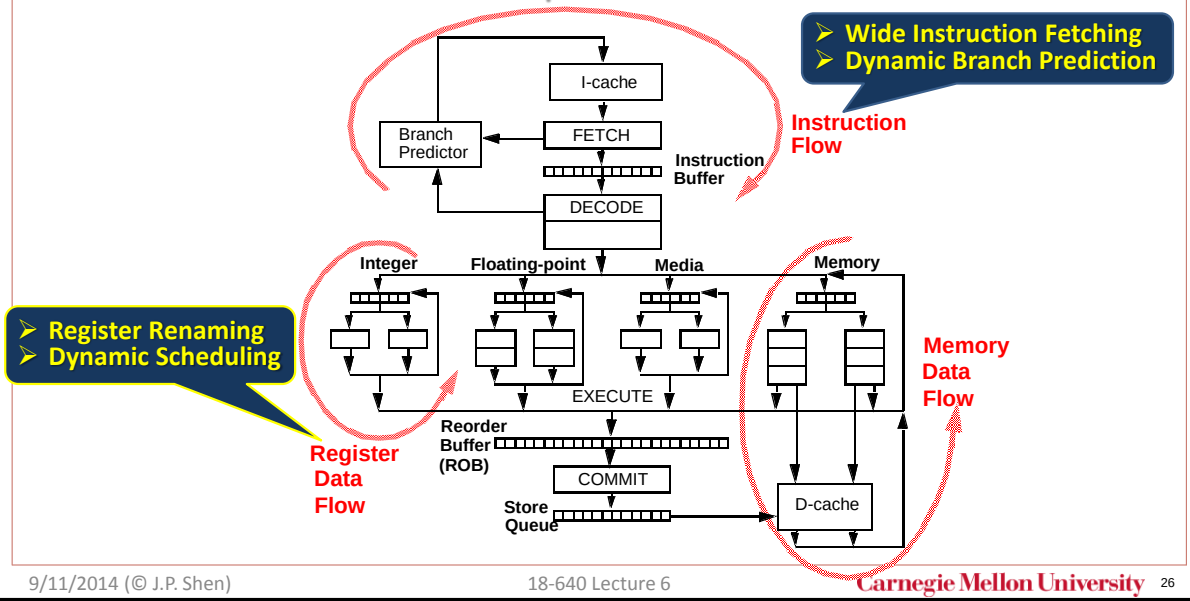
18-640 Foundations of Computer Architecture

Lecture 6:
"Register Data Flow Techniques"

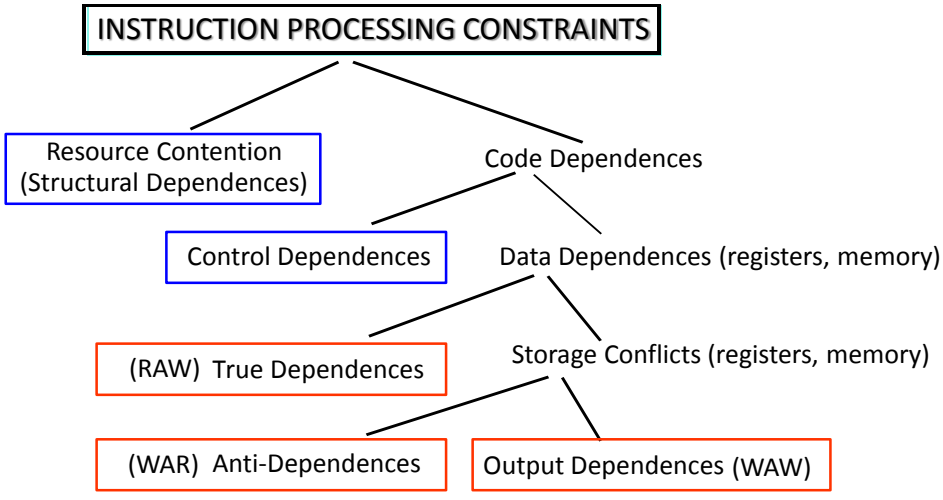
A. Register Data Flow
B. Register Renaming
C. Tomasulo's Algorithm

Electrical & Computer ENGINEERING

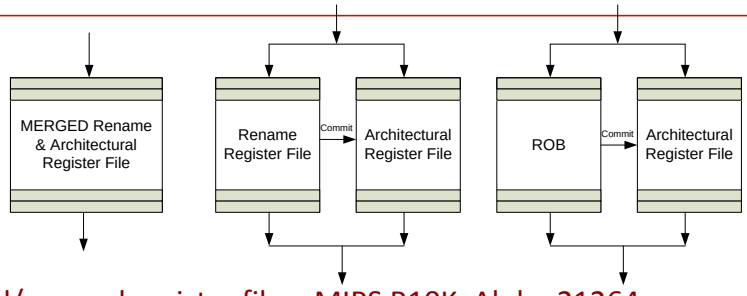
Three Flow Paths of Superscalar Processors



The Big Picture: Impediments Limiting ILP

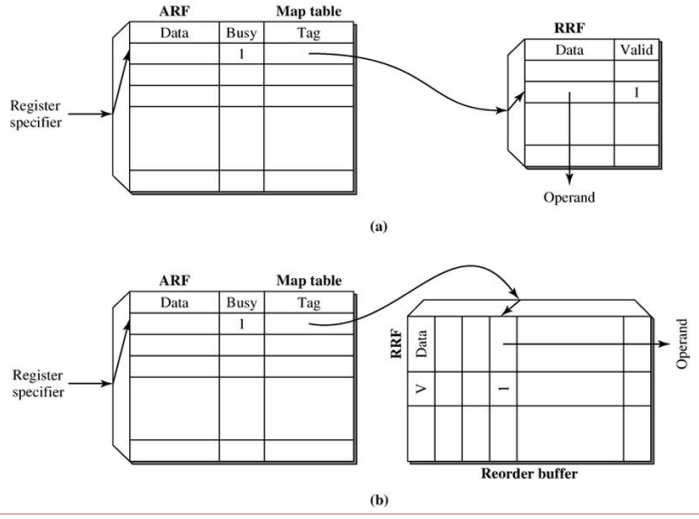


Renaming Buffer Options



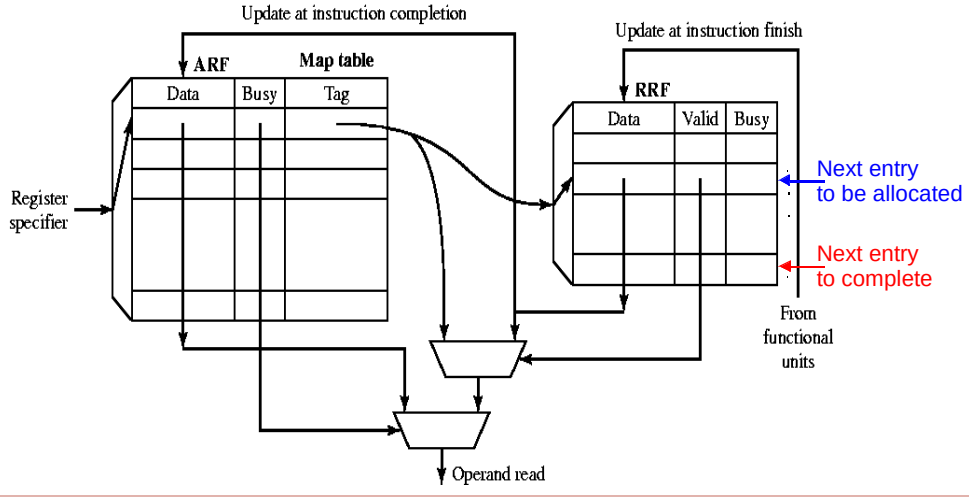
- Unified/merged register file – MIPS R10K, Alpha 21264
  - Registers change role architecture to renamed
- Rename register file (RRF) – PA 8500, PPC 620
  - Holds new values until they are committed to ARF (extra transfer)
- Renaming in the ROB – Pentium III
- Note: can have a single scheme or separate for integer/FP

Integrating Map Tables with the ARF



Register Renaming Tasks

- Source Read, Destination Allocate, Register Update

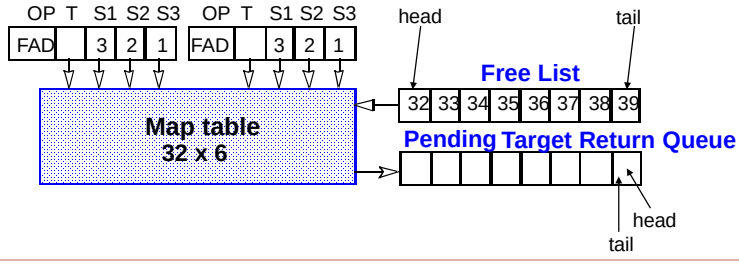


Register Renaming in the IBM RS/6000

Incoming FPU instructions pass through a renaming table prior to decode
Physical register names only within the FPU!!

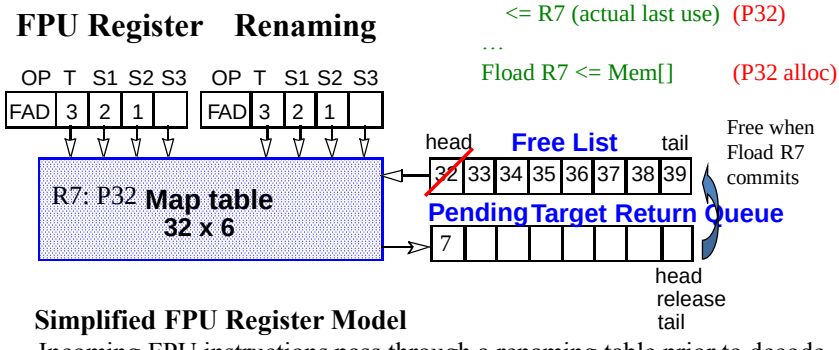
32 architectural registers ⇒ 40 physical registers
Complex control logic maintains active register mapping

FPU Register Renaming



Register Renaming in the IBM RS/6000 FPU

Fload R7 <= Mem[] (P32 freed)
…
<= R7 (actual last use) (P32)
…
Fload R7 <= Mem[] (P32 alloc)



Simplified FPU Register Model

Incoming FPU instructions pass through a renaming table prior to decode
The 32 architectural registers are remapped to 40 physical registers
Physical register names are used within the FPU
Complex control logic maintains active register mapping

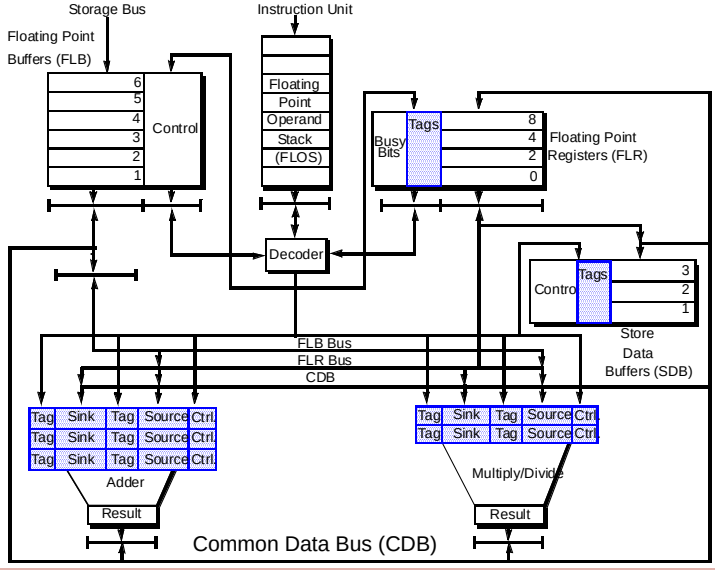
18-640 Foundations of Computer Architecture

Lecture 7:
"Dynamic Scheduling and Out-of-Order Execution"

A. OOO Execution Implementation
   a. Instruction Window Implementation
   b. Reorder Buffer Implementation
B. Dynamic Instruction Scheduling
   a. Instruction Wake Up
   b. Instruction Select
   c. Result Forwarding
C. Intel Pentium Pro Case Study

Electrical & Computer ENGINEERING

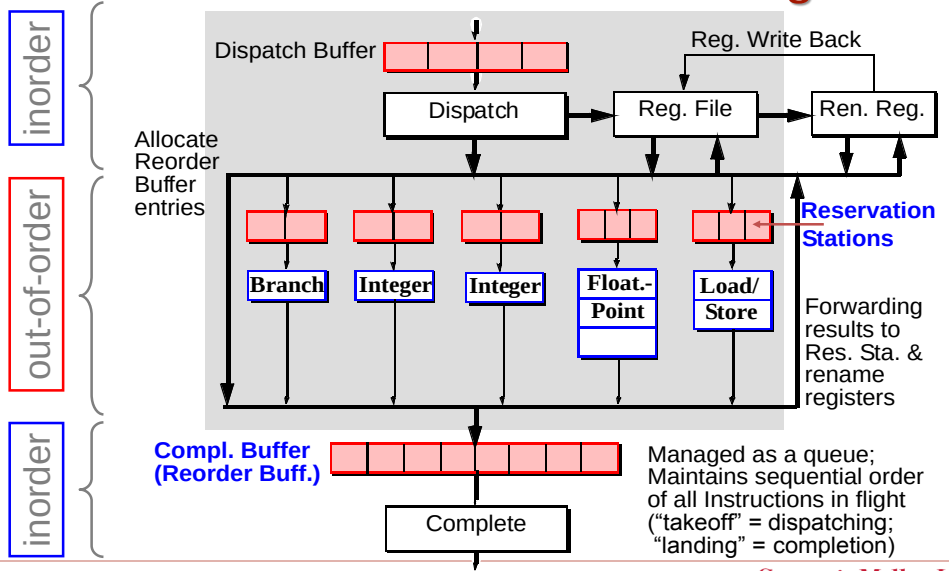
IBM 360/91 FPU (circa 1969)



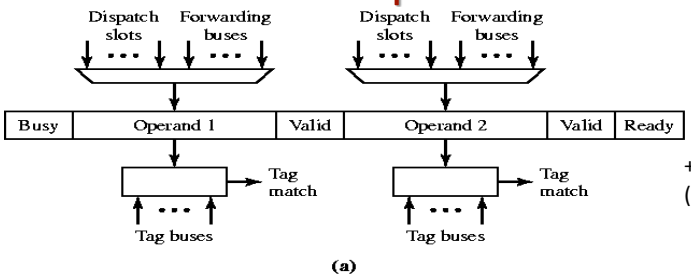
Summary of Tomasulo's Algorithm

- Supports out of order execution of instructions. Resolves dependences dynamically using hardware. Attempts to delay the resolution of dependences as late as possible.
- Structural dependence does not stall issuing; virtual FU's in the form of reservation stations are used.
- Output dependence does not stall issuing; copying of old tag to reservation station and updating of tag field of the register with pending write with the new tag. Can support sequence of multiple output dependences.
- True dependence with a pending write operand does not stall the reading of operands; pseudo operand (tag) is copied to reservation station. Forwarding from FU's to reservation stations bypasses the register file.
- Anti-dependence does not stall write back; earlier copying of operand awaiting read to the reservation station.

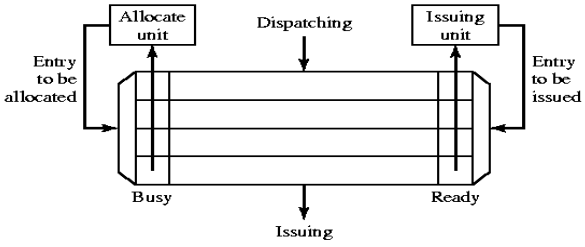
Elements of Modern Micro-Dataflow Engine



Reservation Station Implementation



(a)



(b)

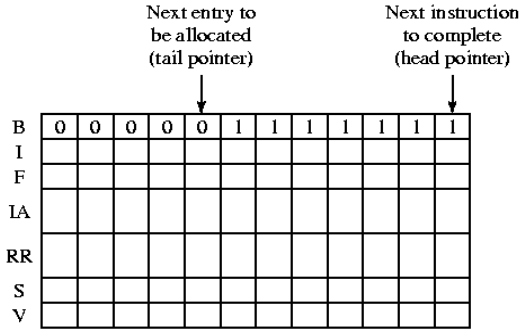
+ info for executing instruction (opcode, ROB entry, RRF entry…)

- Reservation Stations: distributed vs. centralized
  - Wakeup: benefit to partition across data types
  - Select: much easier with partitioned scheme
    - Select 1 of n/4 vs. 4 of n

Reorder Buffer Implementation

Busy	Issued	Finished	Instruction address	Rename register	Speculative	Valid

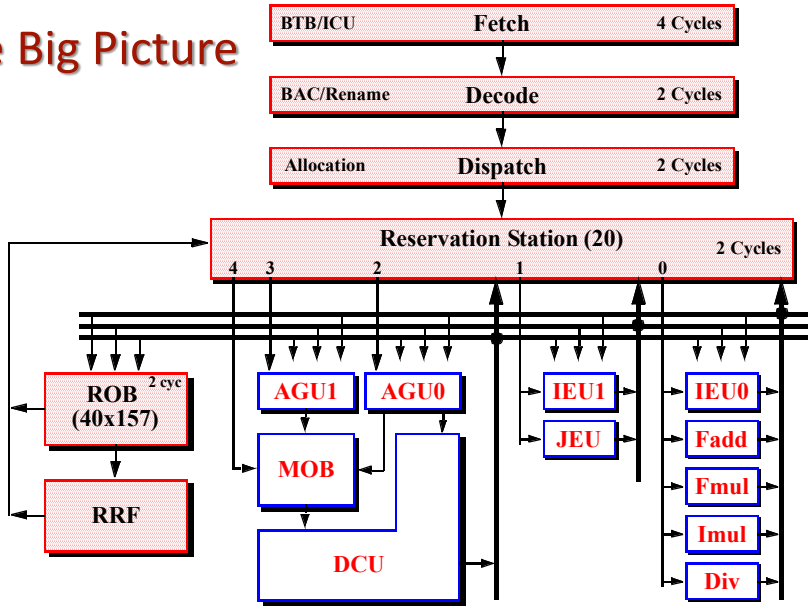
(a)



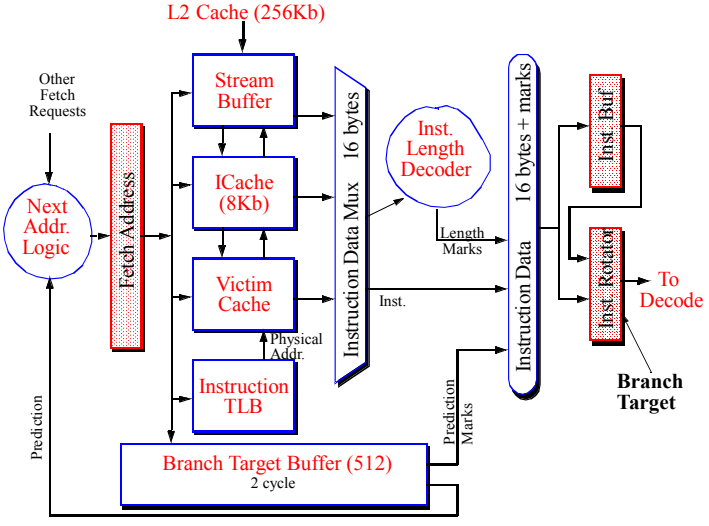
(b)

- Reorder Buffer
  - "Bookkeeping"
  - Can be instruction-grained, or block-grained (4-5 ops)

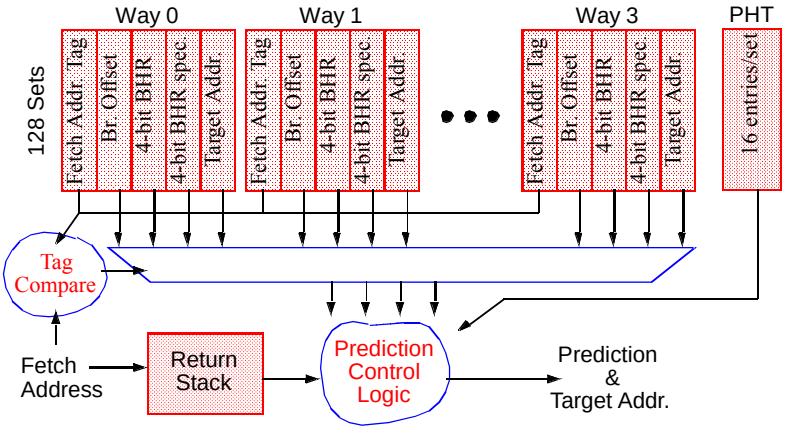
P6 – The Big Picture



Instruction Fetch

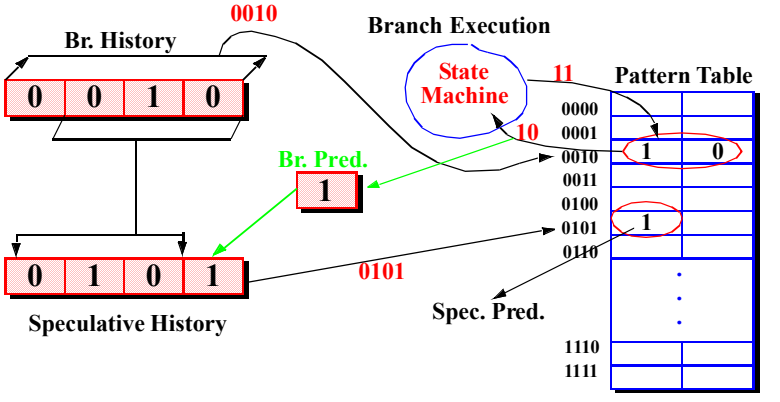


Branch Target Buffer



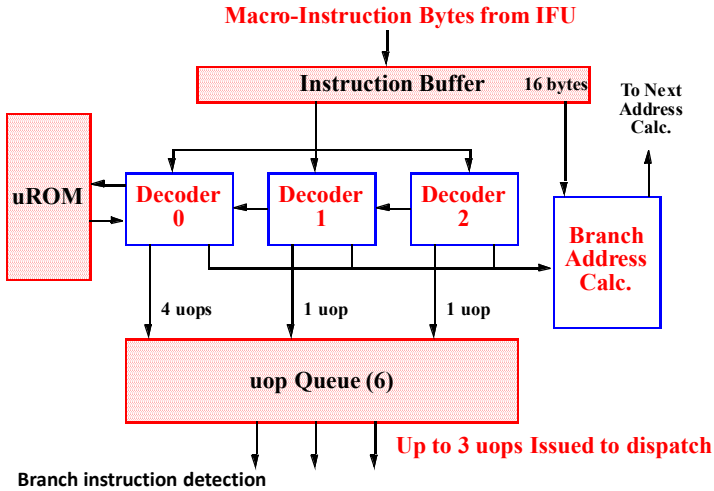
- Pattern History Table (PHT) is not speculatively updated
- A speculative Branch History Register (BHR) and prediction state is maintained
- Uses speculative prediction state if it exist for that branch

Branch Prediction Algorithm



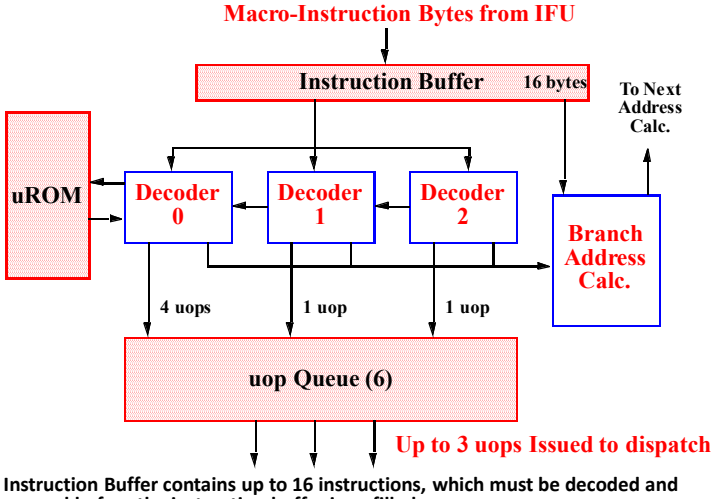
- Current prediction updates the speculative history prior to the next instance of the branch instruction
- Branch History Register (BHR) is updated during branch execution
- Branch recovery flushes front-end and drains the execution core
- Branch mis-prediction resets the speculative branch history state to match BHR

Instruction Decode - 1



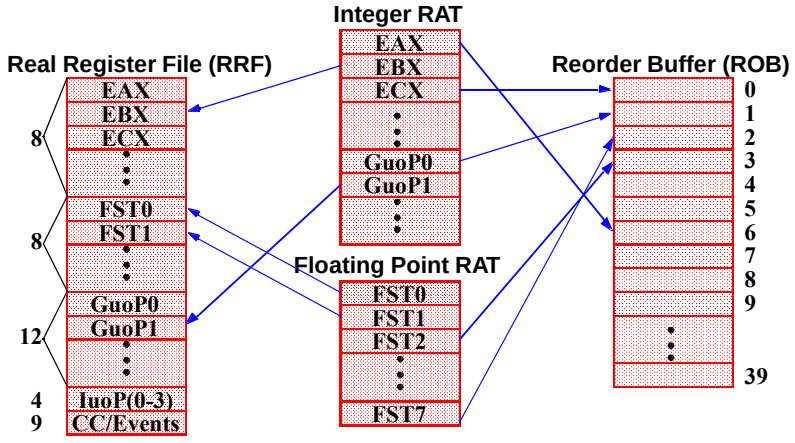
- Branch instruction detection
- Branch address calculation - Static prediction and branch always execution
- One branch decode per cycle (break on branch)

Instruction Decode - 2



- Instruction Buffer contains up to 16 instructions, which must be decoded and queued before the instruction buffer is re-filled
- Macro-instructions must shift from decoder 2 to decoder 1 to decoder 0

Register Renaming - 1

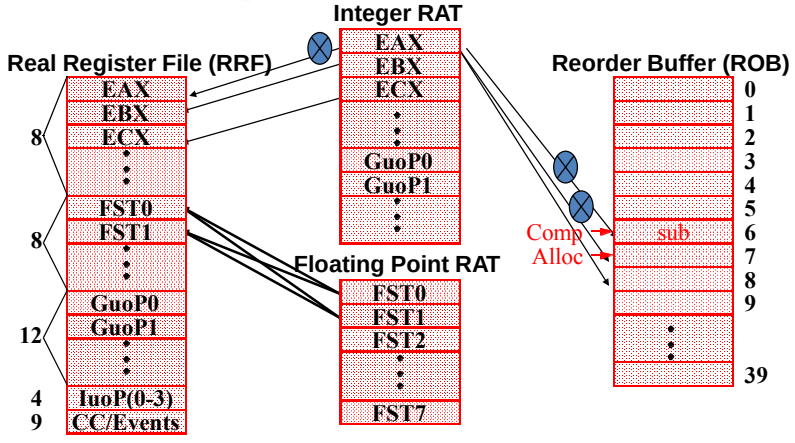


Similar to Tomasulo's Algorithm - Uses ROB entry number as tags

The register alias tables (RAT) maintain a pointer to the most recent data for the renamed register

Execution results are stored in the ROB

Register Renaming - Example



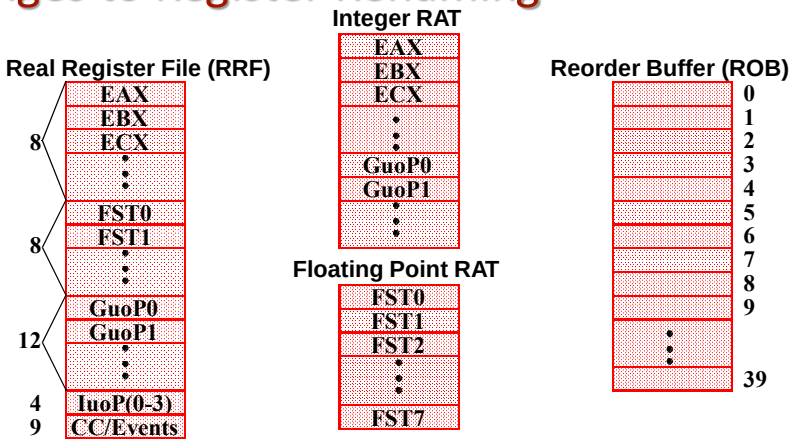
Dispatching:

```
add     eax, ebx
add     eax, ecx
fxch    f0, f1
```

Completing:

```
sub     eax, ecx
```

Challenges to Register Renaming

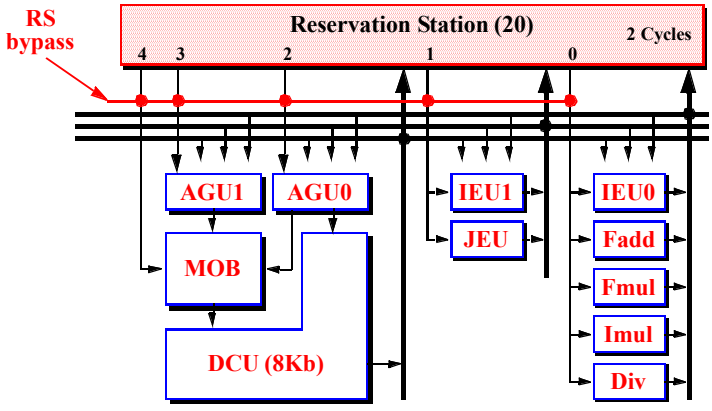


8-bit code

```
mov   AL, #data1
mov   AH, #data2
add   AL, #data3
add   AL, #data4
```

Byte addressable registers

Out-of-Order Execution Engine



- In-order branch issue and execution
- In-order load/store issue to address generation units
- Instruction execution and result bus scheduling
- Is the reservation station *"truly"* centralized & what is *"binding"*?

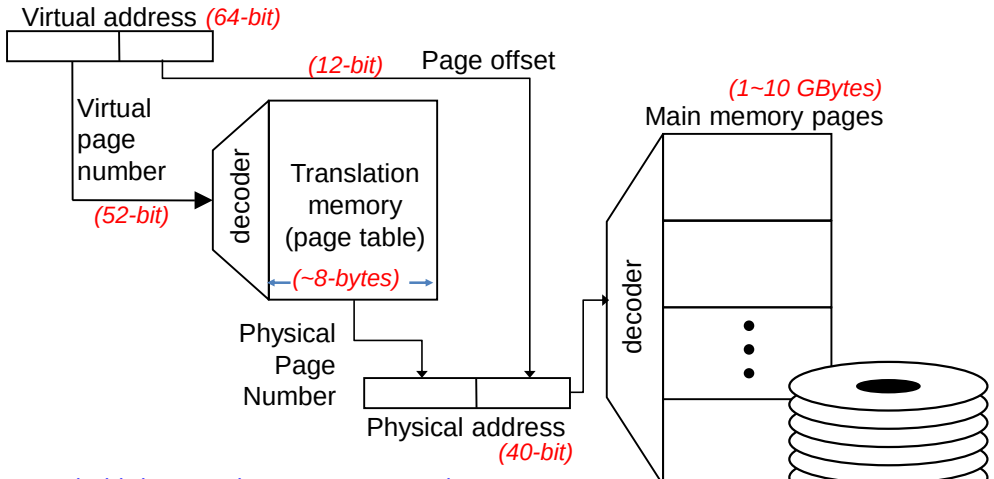
18-640 Foundations of Computer Architecture

Lecture 8:
"Memory Data Flow Techniques"

A. Memory Hierarchy Revisited
B. Virtual Memory Revisited
C. Memory Data Flow Techniques
   a. Memory Data Dependences
   b. Load Bypassing
   c. Load Forwarding
   d. Speculative Disambiguation
   e. The Memory Bottleneck

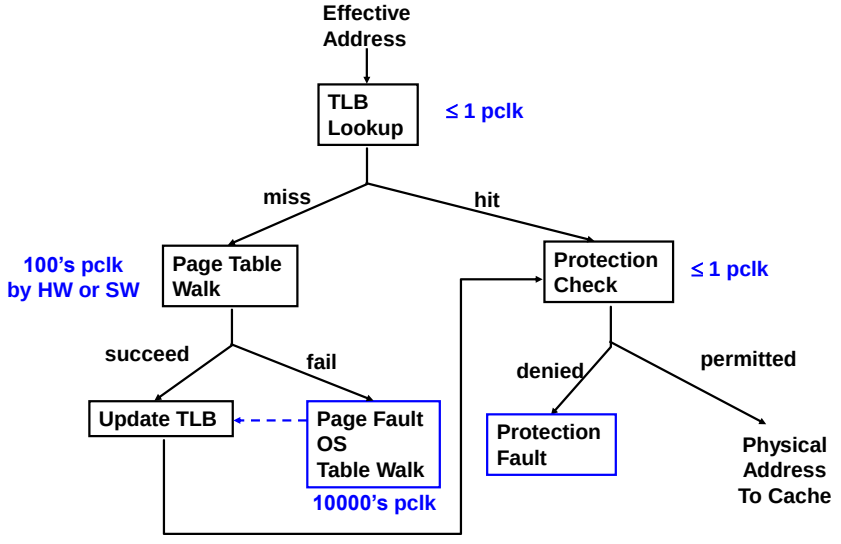
Electrical & Computer ENGINEERING

Page-Based Virtual Memory

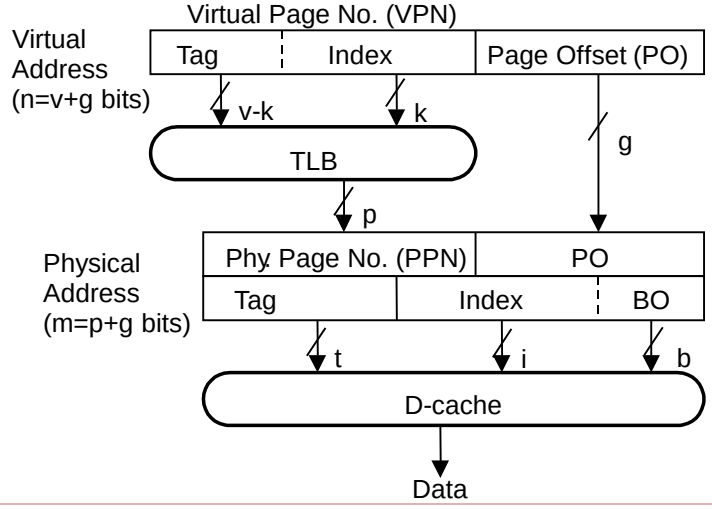


Where to hold this translation memory and how much translation memory do we need?

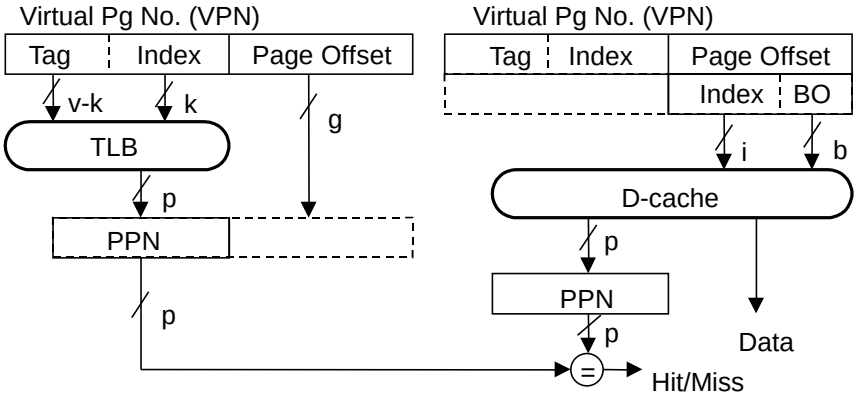
Virtual to Physical Address Translation



Physically Indexed Cache

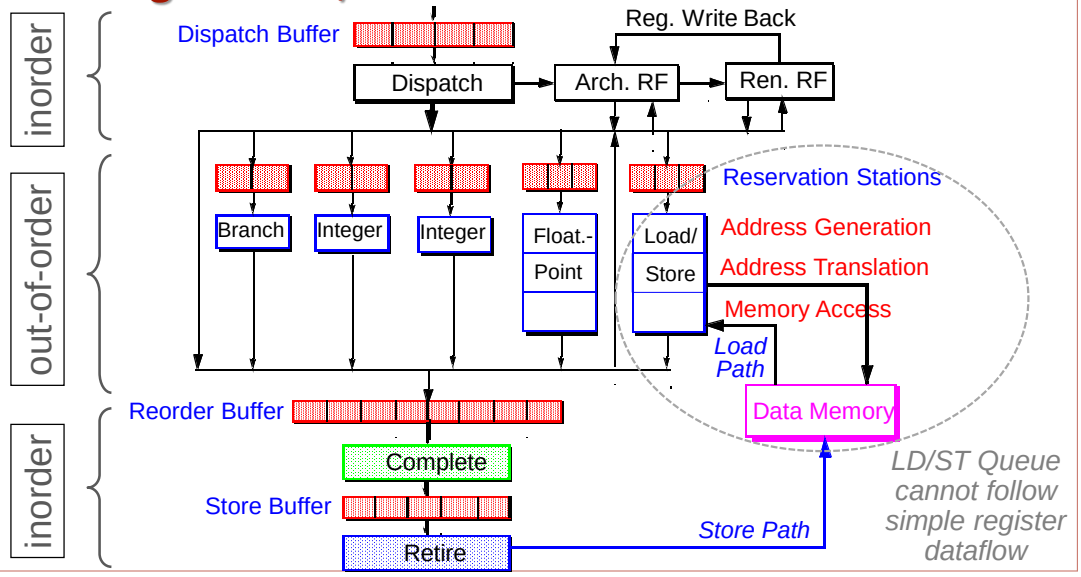


Virtually Indexed Cache



How big can a virtually indexed cache get?

Processing of Load/Store Instructions



Load/Store Units and Store Buffer

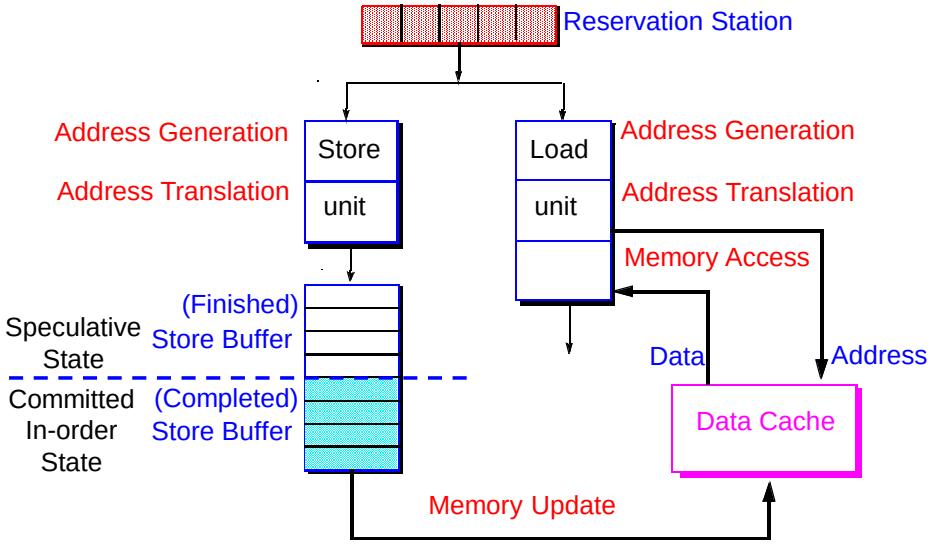


Illustration of Load Bypassing

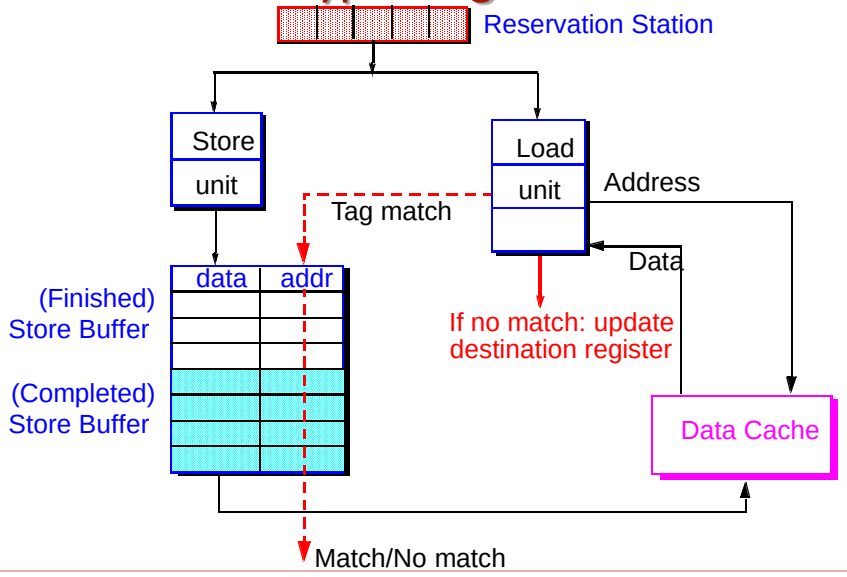
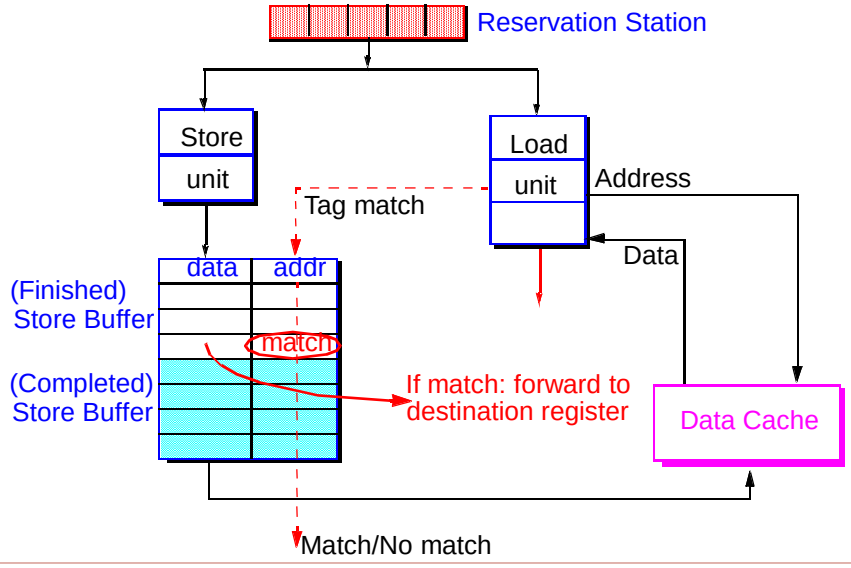
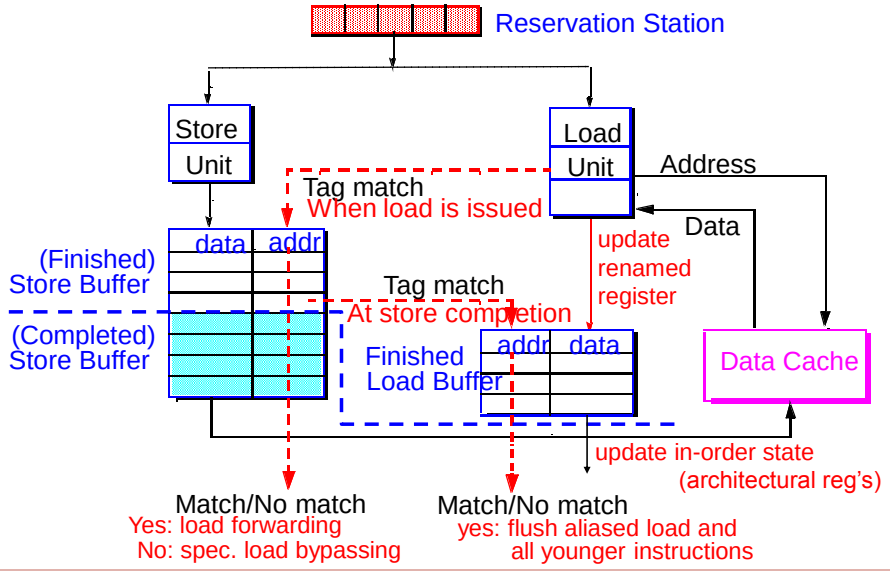


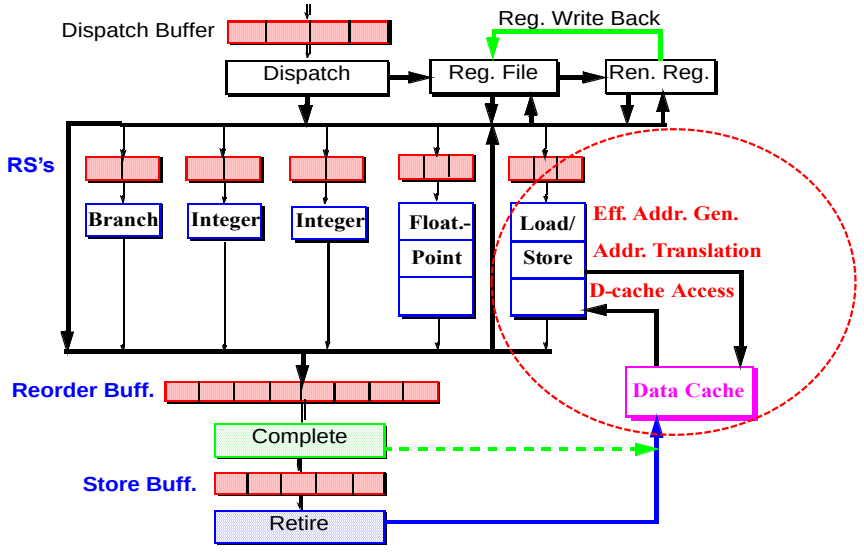
Illustration of Load Forwarding



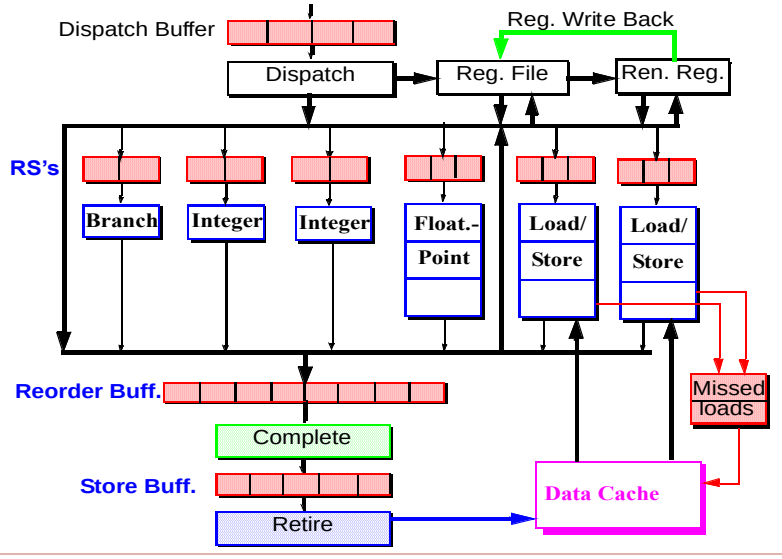
Speculative Load Bypassing



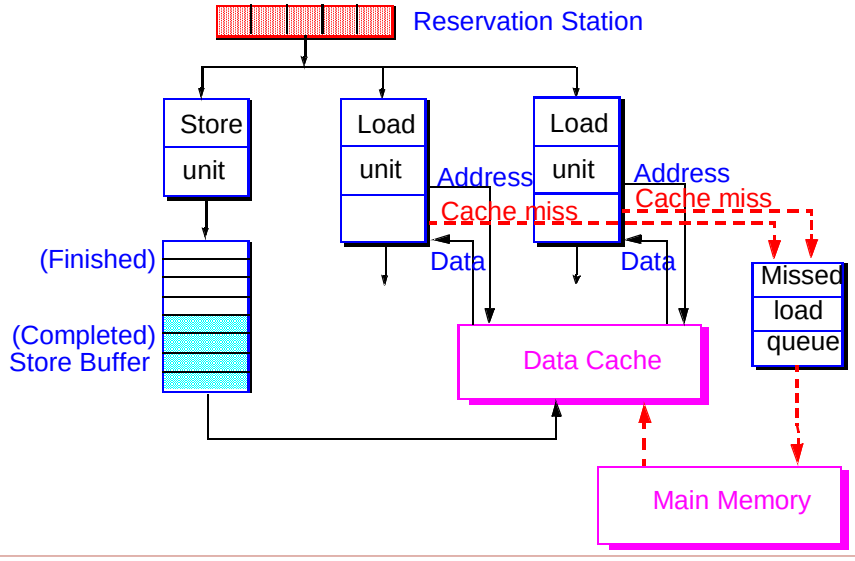
C.e. The Memory Bottleneck



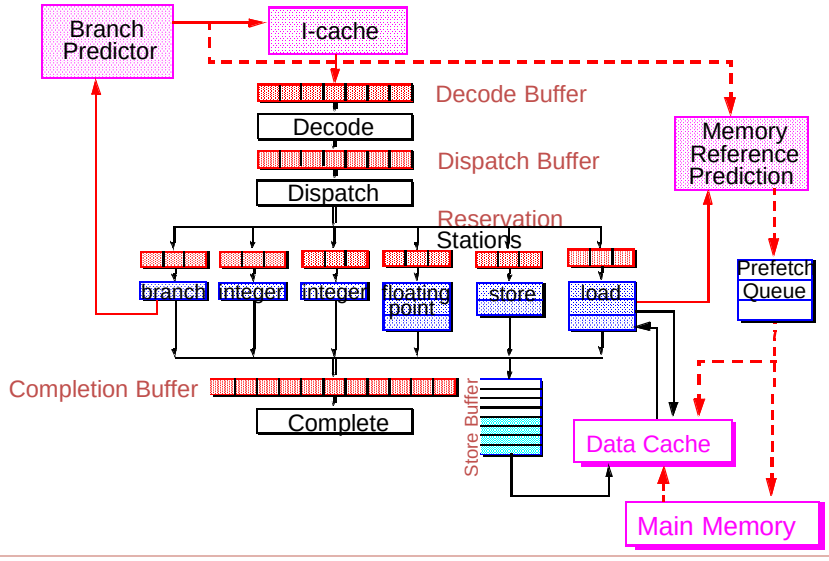
Easing the Memory Bottleneck (missed-load buffer)



Dual-Ported Non-Blocking Cache



Prefetching Data Cache



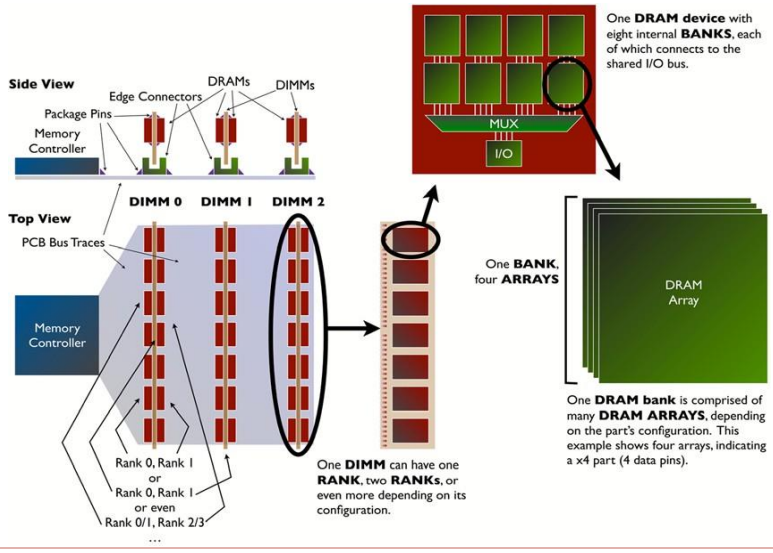
18-640 Foundations of Computer Architecture

Lecture 9:
"Main Memory System Design"

A. Main Memory Implementation
B. DRAM Organization
C. DRAM Operation
D. Memory Controller
E. Emerging Technologies

Electrical & Computer ENGINEERING

A. Main Memory Implementation

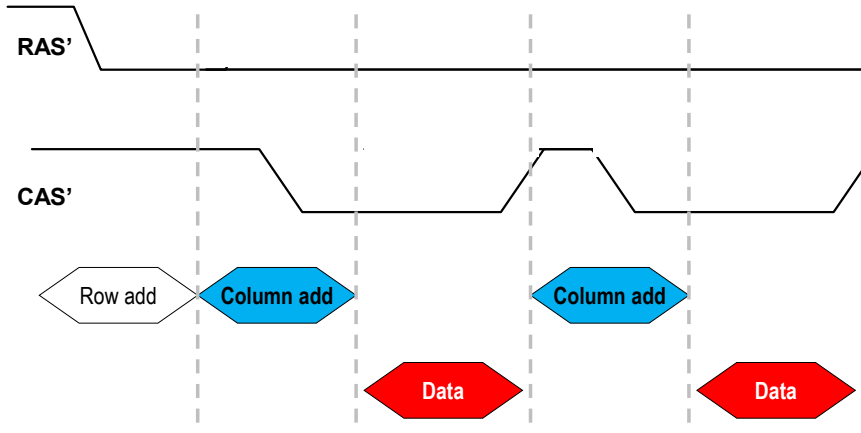


Page Mode DRAM

- A DRAM bank is a 2D array of cells: rows x columns
- A "DRAM row" is also called a "DRAM page"
- "Sense amplifiers" also called "row buffer"

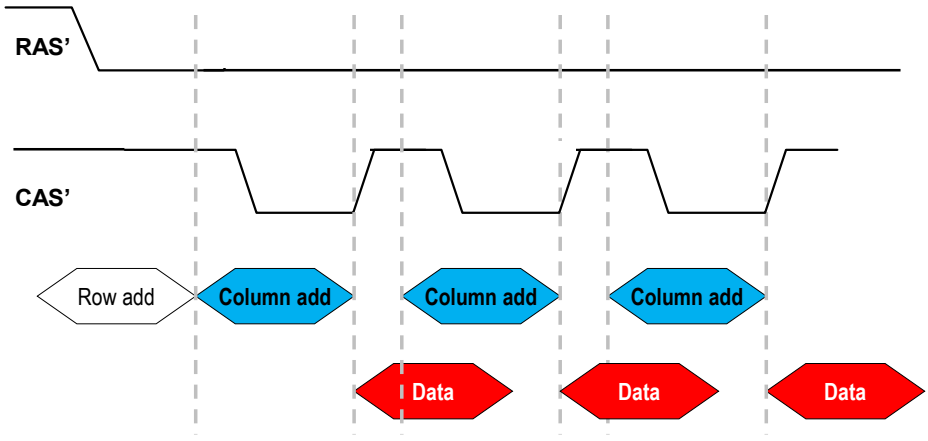
- Each address is a <row,column> pair
- Access to a "closed row"
  - Activate command opens row (placed into row buffer)
  - Read/write command reads/writes column in the row buffer
  - Precharge command closes the row and prepares the bank for next access
- Access to an "open row"
  - No need for activate command

Fast Page Mode (FPM)



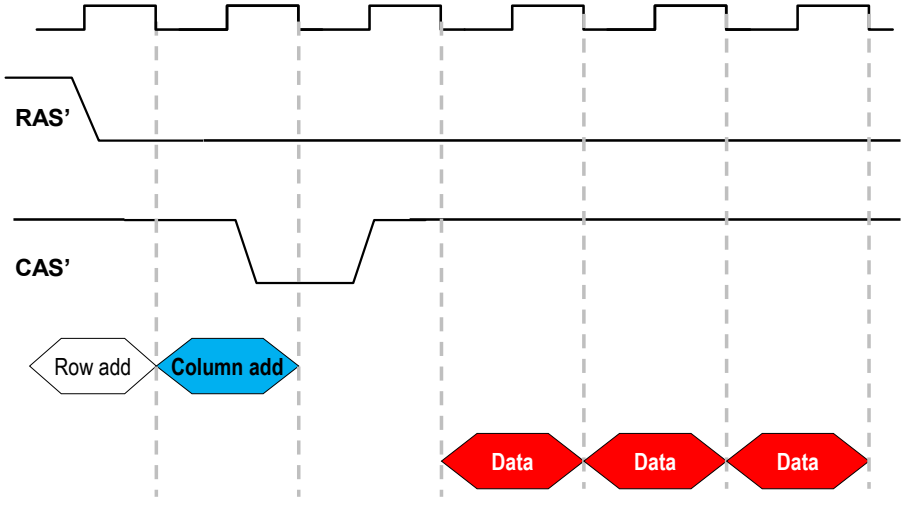
- One row address
- Multiple column addresses

Extended Data Out (EDO)



- As in FPM
- But overlapped Column Address assert with Data Out

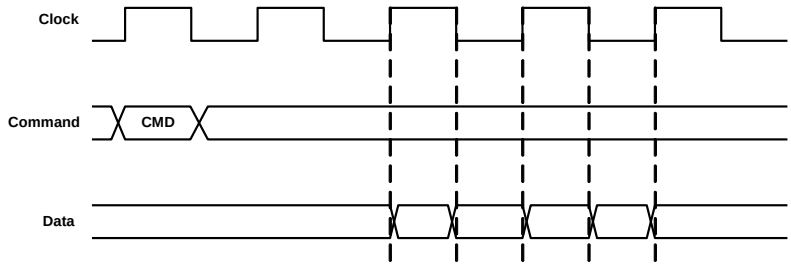
Synchronous DRAM (SDRAM)



- Single CAS Strobe, multiple transfers

DDR SDRAM Timing

- **Read access**



18-640 Foundations of Computer Architecture

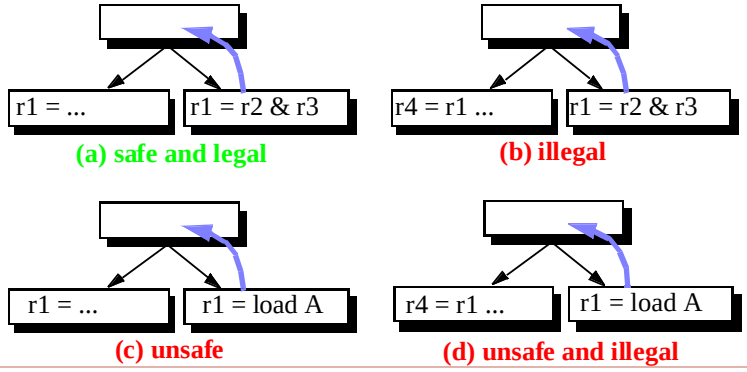
Lecture 10: "VLIW and EPIC Architectures"

A. Code Scheduling
B. Trace Scheduling
C. VLIW Architecture
D. Multiflow TRACE Computer
E. Intel IA-64 EPIC Architecture

Electrical & Computer ENGINEERING

Types of Speculative Code Motion

- Two characteristics of speculative code motion:
  - safety, which indicates whether or not spurious exceptions may occur
  - legality, which indicates correctness of results
- Four possible types of code motion:



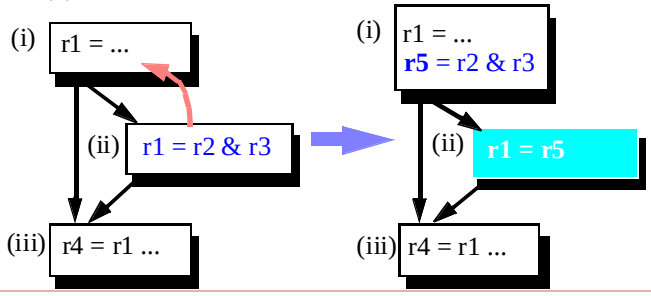
Register Renaming

- Prevents boosted instructions from overwriting register state needed on alternate execution path.
- Utilizes idle (non-live) registers (r6 in example below).

BB#	Original Code			Scheduled Code		
n		load	r4= ...		load	r4= ...
		load	r5= ...		load	r5= ...
		cmpi	c0,r4,10		cmpi	c0,r4,10
		add	r4=r4+r5		add	r4=r4+r5
		<stall>			sub	r3=r7-r4
		<stall>			and	**r6**=r3&r5
		bc	c0, A1		bc	c0, A1
n+1		st	... =r4		st	... =r4
n+2	A1:	sub	r3=r7-r4	A1:	st	... =**r6**
		and	r4=r3&r5			
		st	... =r4			

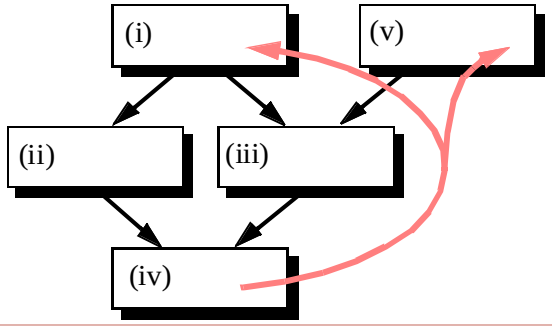
Copy Creation

- Register renaming causes a problem when there are multiple definitions of a register reaching a single use:
  - Below, definitions of r1 in both (i) and (ii) reach the use in (iii).
  - If the instruction in (ii) is boosted into (i), it must be renamed to preserve the first value of r1.
  - However, the boosted definition of r1 must reach the use in (iii) as well.
  - Hence, we insert a copy instruction in (ii).



Instruction Replication

- General case of upward code motion: crossing control flow joins.
- Instructions must be present on each control flow path to their original basic block
- Replicate set is computed for each basic block that is a source for instructions to be boosted

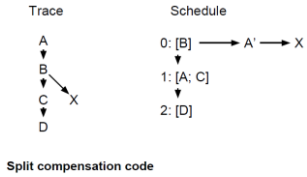


Trace Scheduling Overview

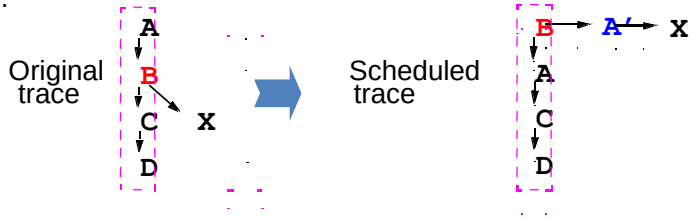
- Trace Selection
  - Select seed (the highest frequency basic block)
  - Extend trace (along the highest frequency edges)
    - forward (successor of the last block of the trace)
    - backward (predecessor of the first block of the trace)
  - Don't cross loop back edge
  - Bound max_trace_length heuristically
- Trace Scheduling
  - Build data precedence graph for a whole trace
  - Perform list scheduling and allocate registers
  - Add compensation code to maintain semantic correctness
- Speculative Code Motion (upward)
  - Move an instruction above branches if safe

Compensation Code for Code Motion

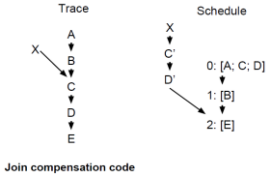
- Split Compensation Code:
  - Instruction with more than one successor



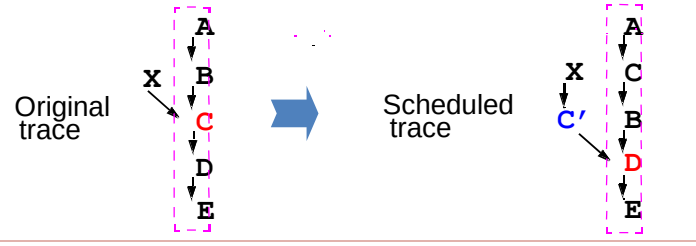
Split compensation code



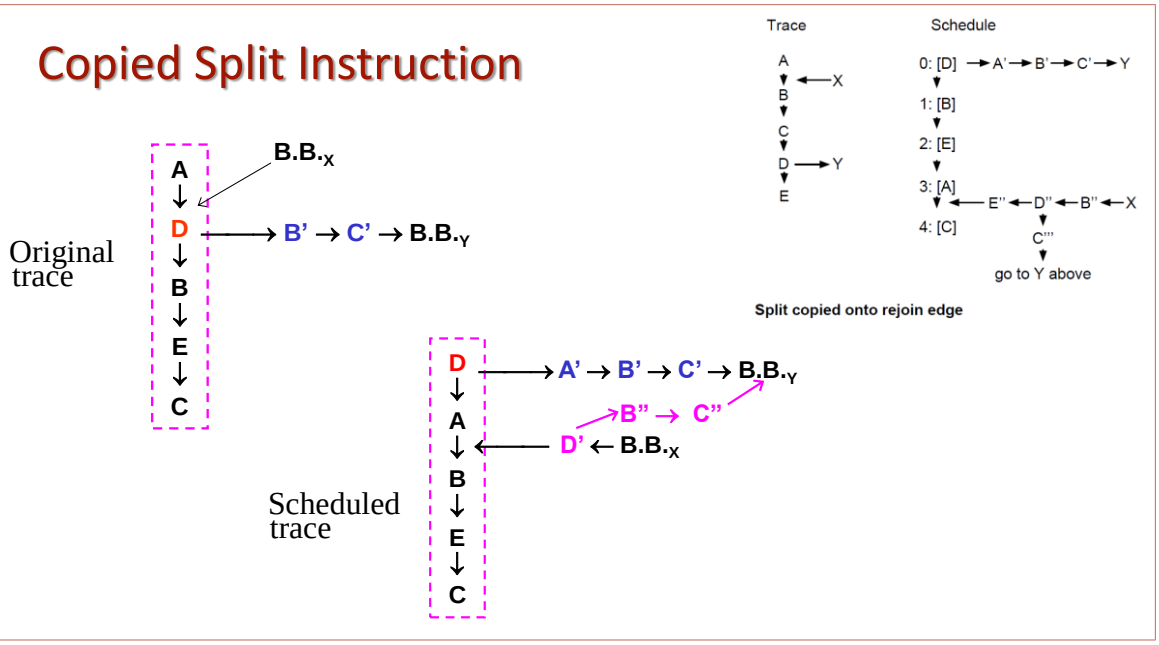
- Join Compensation Code:
  - Instruction with more than one predecessor



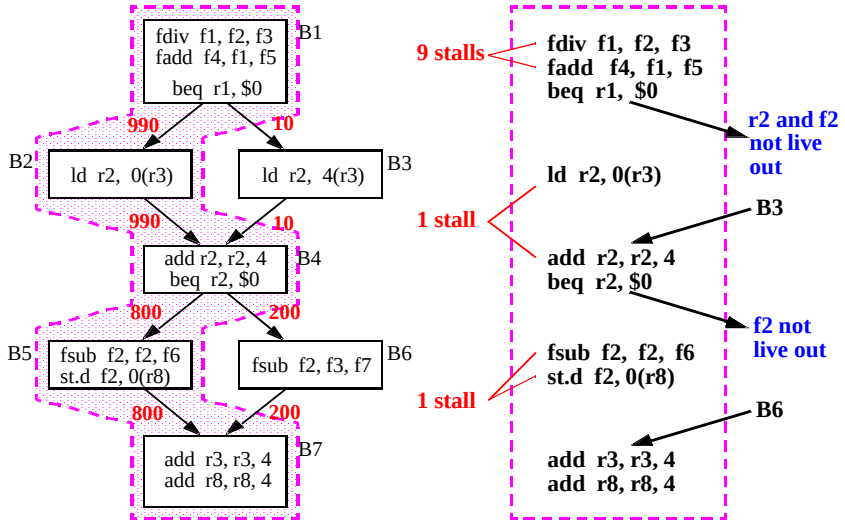
Join compensation code



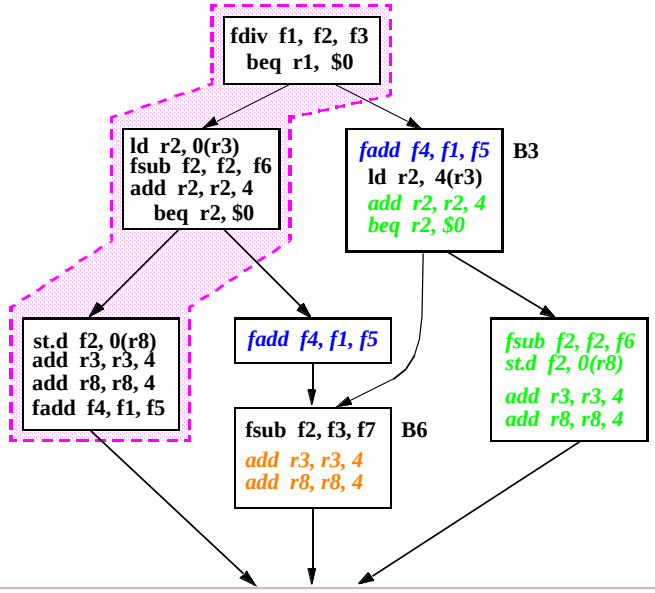
Copied Split Instruction



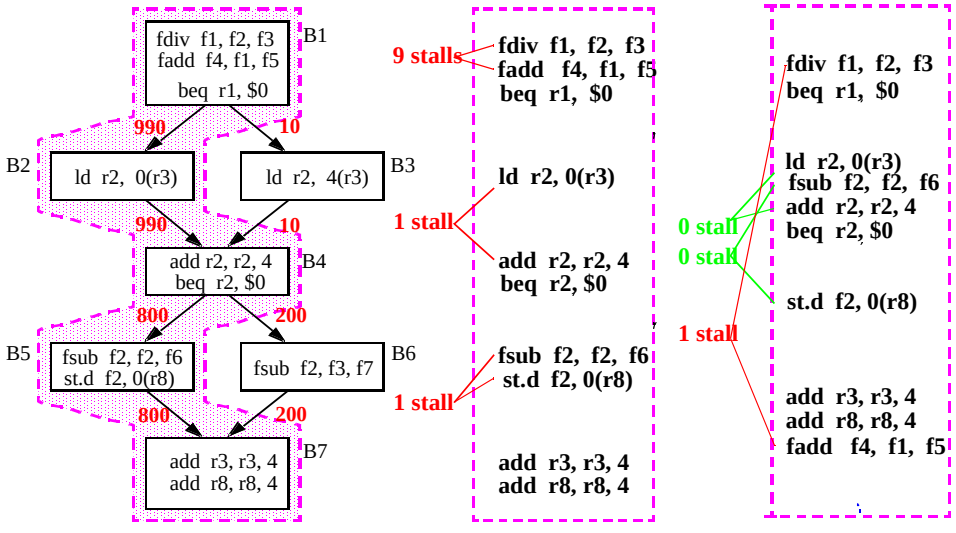
Trace Scheduling Example



Compensation Code Illustration



Trace Scheduling Performance Improvement



Strengths of VLIW Technology

- Parallelism can be exploited at the instruction level
  - Available in both vectorizable and sequential programs
- Hardware is regular and straightforward
  - Most hardware is in the datapath performing useful computations
  - Instruction issue costs scale approximately linearly

  Potentially very high clock rate
- Architecture is "*Compiler Friendly*"
  - Implementation is completely exposed - 0 layer of interpretation
  - Compile time information is easily propagated to run time
- Exceptions and interrupts are easily managed
- Run-time behavior is highly predictable
  - Allows real-time applications
  - Greater potential for code optimization

Weaknesses of VLIW Technology

- No object code compatibility between generations
- Program size is large (explicit NOPs)

  Multiflow machines predated "dynamic memory compression" by encoding NOPs in the instruction memory
- Compilers are extremely complex
  - Assembly code is almost impossible
- Philosophically incompatible with caching techniques
- VLIW memory systems can be very complex
  - Simple memory systems may provide very low performance
  - Program controlled multi-layer, multi-banked memory
- Parallelism is underutilized for some algorithms

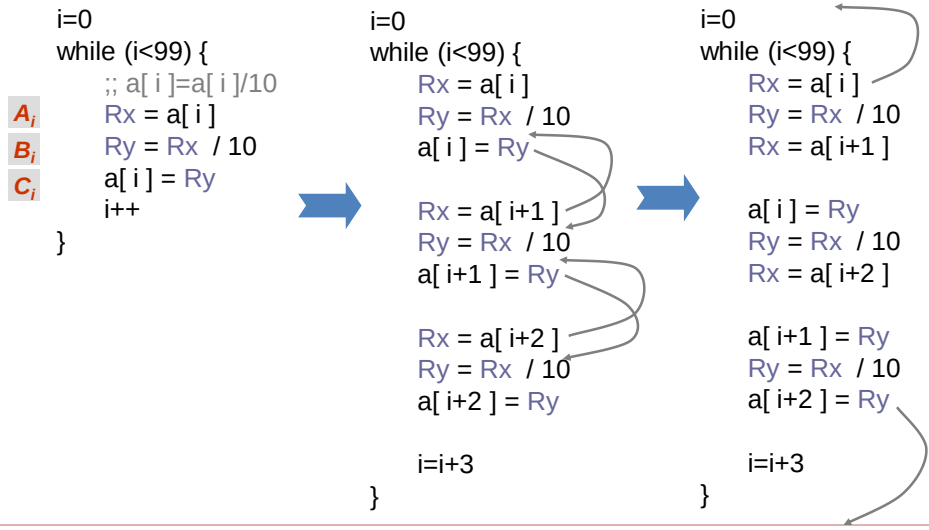
Loop Unrolling

```
i = 1;
while ( i < 100 ) {
    a[i] = b[i+1] + (i+1)/m
    b[i] = a[i-1] - i/m
    i = i + 1
}
```

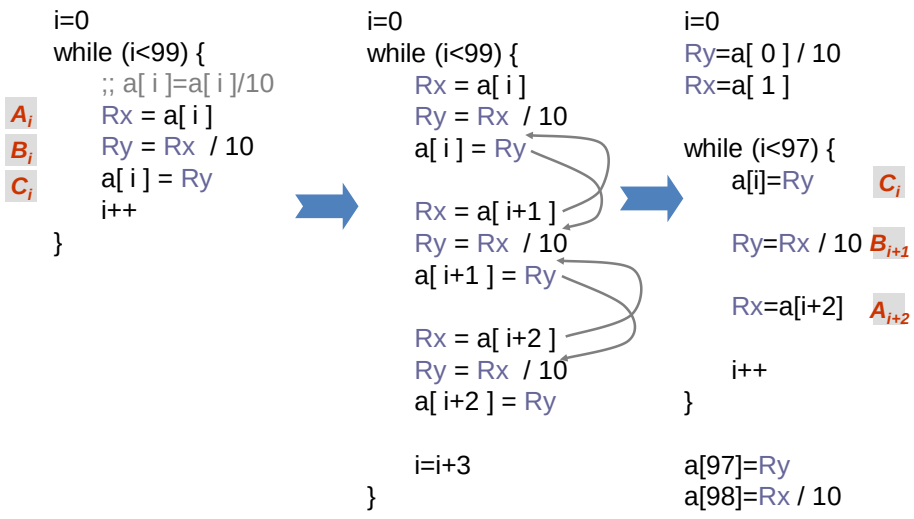
```
i = 1;
while ( i < 100 ) {
    a[i] = b[i+1] + (i+1)/m
    b[i] = a[i-1] - i/m
    a[i+1] = b[i+2] + (i+2)/m
    b[i+1] = a[i] - (i+1)/m
    i = i + 2
}
```

- Reduce loop overhead
  - Increment induction variable
  - Loop condition test
- Enlarged basic block (and analysis scope)
  - Instruction-level parallelism
  - More common subexpression
  - Memory accesses (aggressive memory aliasing analysis)

Software Pipelining



Software Pipelining *(continued)*



IA-64 EPIC vs. Classic VLIW

- Similarities:
  - Compiler generated wide instructions
  - Static detection of dependences
  - ILP encoded in the binary (a group)
  - Large number of architected registers
- Differences:
  - Instructions in a bundle can have dependences
  - Hardware interlock between dependent instructions
  - Accommodates varying number of functional units and latencies
  - Allows dynamic scheduling and functional unit binding

  Static scheduling are "suggestive" rather than absolute

  ⇒Code compatibility across generations

  but software won't run at top speed until it is recompiled so "shrink-wrap binary" might need to include multiple builds

E. Intel IA-64 (Itanium) Architecture

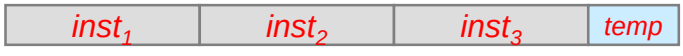
- 128 general-purpose registers
- 128 floating-point registers
- Arbitrary number of functional units
- Arbitrary latencies on the functional units
- Arbitrary number of memory ports
- Arbitrary implementation of the memory hierarchy

- *Related to but not the same as VLIW!!*
- *Binary compatible, but needs retargetable compiler and recompilation to achieve maximum program performance on different IA-64 implementations,*

IA-64 Instruction Format

- IA-64 "Bundle"
  - Total of 128 bits
  - Contains three IA-64 instructions *(aka syllables)*
  - Template bits in each bundle specify dependences both within a bundle as well as between sequential bundles
  - A collection of independent bundles forms a "group"

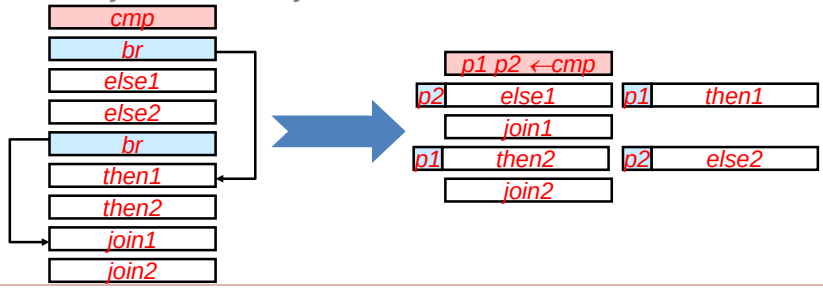
    A more efficient and flexible way to encode ILP then a fixed VLIW format



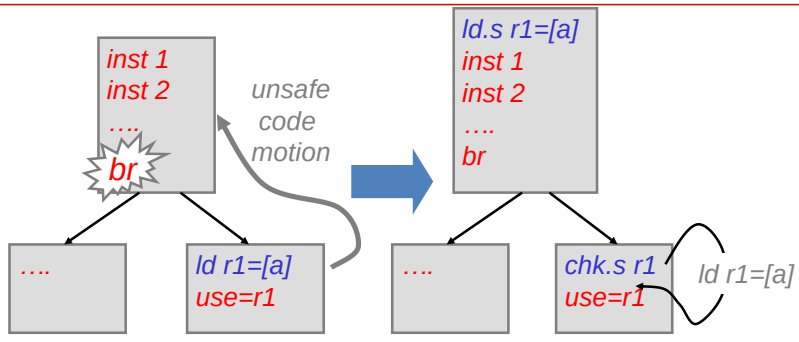
- IA-64 Instruction
  - Fixed-length 40 bits long
  - Contains three 7-bit register specifiers
  - Contains a 6-bit field for specifying one of the 64 one-bit predicate registers

Predicated Execution

- Each instruction can be separately predicated
- 64 one-bit predicate registers
  - *Each instruction carries a 6-bit predicate field*
- An instruction is effectively a NOP if its predicate is false
- Assumes IA-64 processors have lots of spare resources
- *Converts control flow into dataflow*

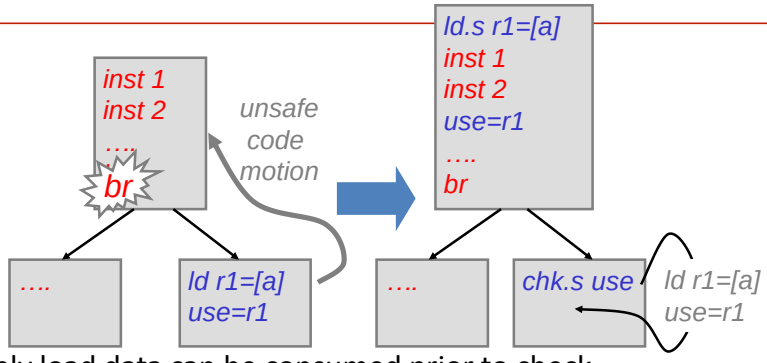


Speculative Non-Faulting Load



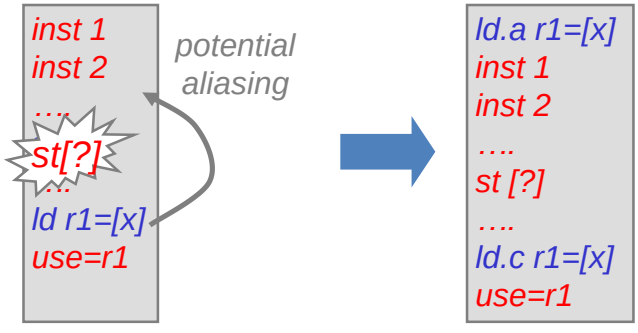
- *ld.s* fetches *speculatively* from memory
  i.e. any exception due to *ld.s* is suppressed
- If *ld.s r* did not cause an exception then *chk.s r* is an NOP, else a branch is taken (to some compensation code)

Speculative, Non-Faulting Load



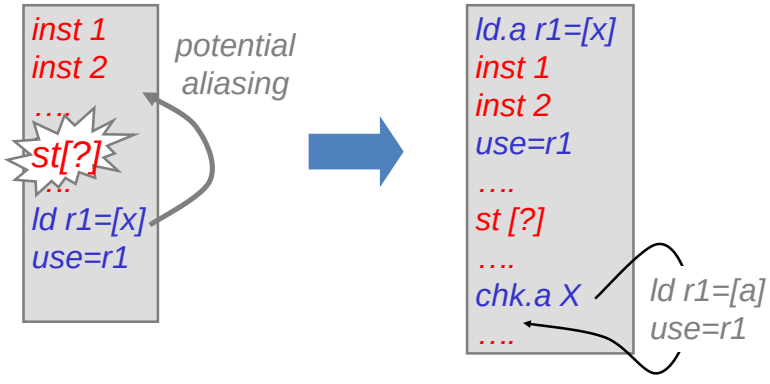
- Speculatively load data can be consumed prior to check
- "speculation" status is propagated with speculated data
- Any instruction that uses a speculative result also becomes speculative itself (i.e. suppressed exceptions)
- *chk.s* checks the entire dataflow sequence for exceptions

Speculative "Advanced" Load

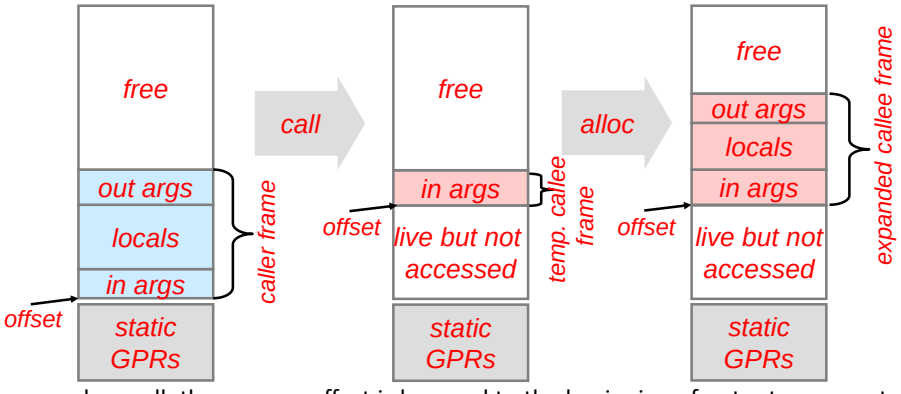


- *ld.a* starts the monitoring of any store to the same address as the advanced load
- If no aliasing has occurred since *ld.a*, *ld.c* is a NOP
- If aliasing has occurred, *ld.c* re-loads from memory

Using Speculative Load Results



Register Stack for Procedure Calls



- On a procedure call, the rename offset is bumped to the beginning of output argument registers
- Callee can then allocate its own working frame (up to 96 regs)
- If there isn't enough free regs to be allocated, HW automatically frees up space by *spilling* live contents not in the current frame to memory *Register stack appears infinite to SW*

18-640 Foundations of Computer Architecture

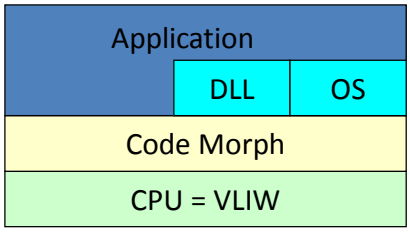
Lecture 11:
"Dynamic Code Translation & Compilation"

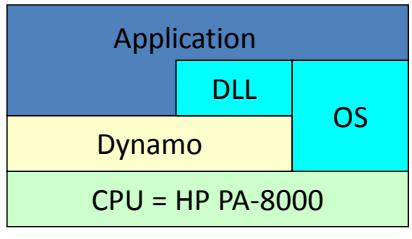
A. Dynamic Binary Translation
B. Transmeta Example
C. Code Morphing System
D. Nvidia Denver Example

Electrical & Computer ENGINEERING

DBT Configurations (1)

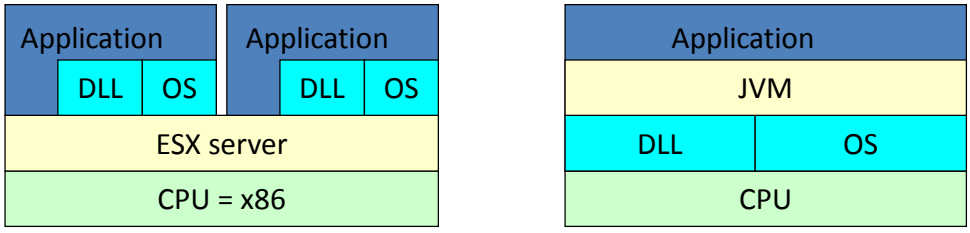
- Cross platform
  - E.g. Crusoe (Transmeta)
- Same platform
  - E.g. Dynamo (HP), PIN (x86), Dynamo-Rio(x86)

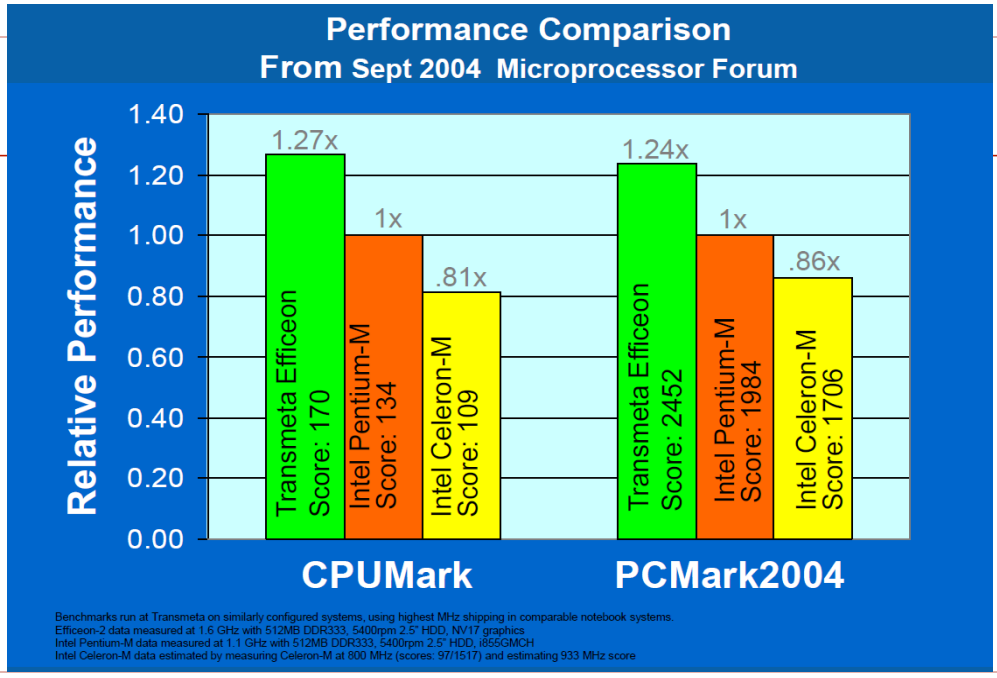




DBT Configurations (2)

- Virtual Machine
  - E.g. ESX server (vmWare)
- JIT compilation
  - E.g. JVM (Sun), C#(MS)





Translation Example

- translation example
  - X86 instruction

```
A. addl %eax,(%esp) // load data from stack, add to %eax
B. addl %ebx,(%esp) // ditto, for %ebx
C. movl %esi,(%ebp) // load %esi from memory
D. subl %ecx,5 // subtract 5 from %ecx registe
```

  - In a first pass, the front end of the translation – simple translation

```
ld %r30,[%esp] // load from stack, into temporary
add.c %eax,%eax,%r30 // add to %eax, set condition codes.
ld %r31,[%esp]
add.c %ebx,%ebx,%r31
ld %esi,[%ebp]
sub.c %ecx,%ecx,5
```

  - In a second pass, the optimizer. applying well known compiler optimization skill such as common subexpression elimination, loop invariant removal or dead code elimination.

```
ld %r30,[%esp] // load from stack only once
add %eax,%eax,%r30
add %ebx,%ebx,%r30 // reuse data loaded earlier
ld %esi,[%ebp]
sub.c %ecx,%ecx,5 // only this last condition code needed
```

  - In a final pass, the scheduler. reordering atoms into molecules.

```
1. ld %r30,[%esp]; sub.c %ecx,%ecx,5
2. ld %esi,[%ebp]; add %eax,%eax,%r30; add %ebx,%ebx,%r30
```


D. NVIDIA "Denver" Architecture

- 128 Mb translation cache in memory
- VLIW engine