



# Microcoded and VLIW Processors

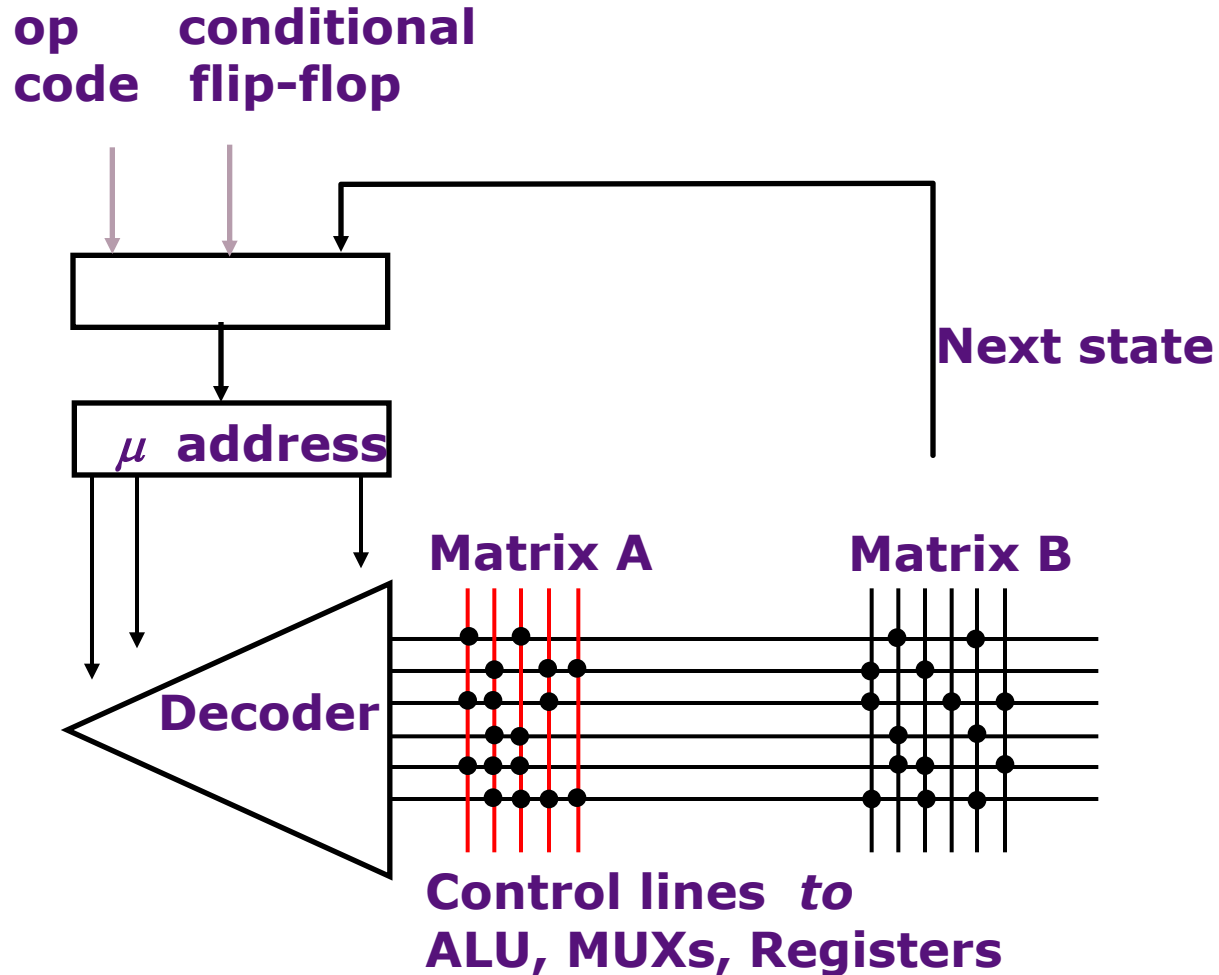
*Joel Emer*

Computer Science & Artificial Intelligence Lab  
M.I.T.

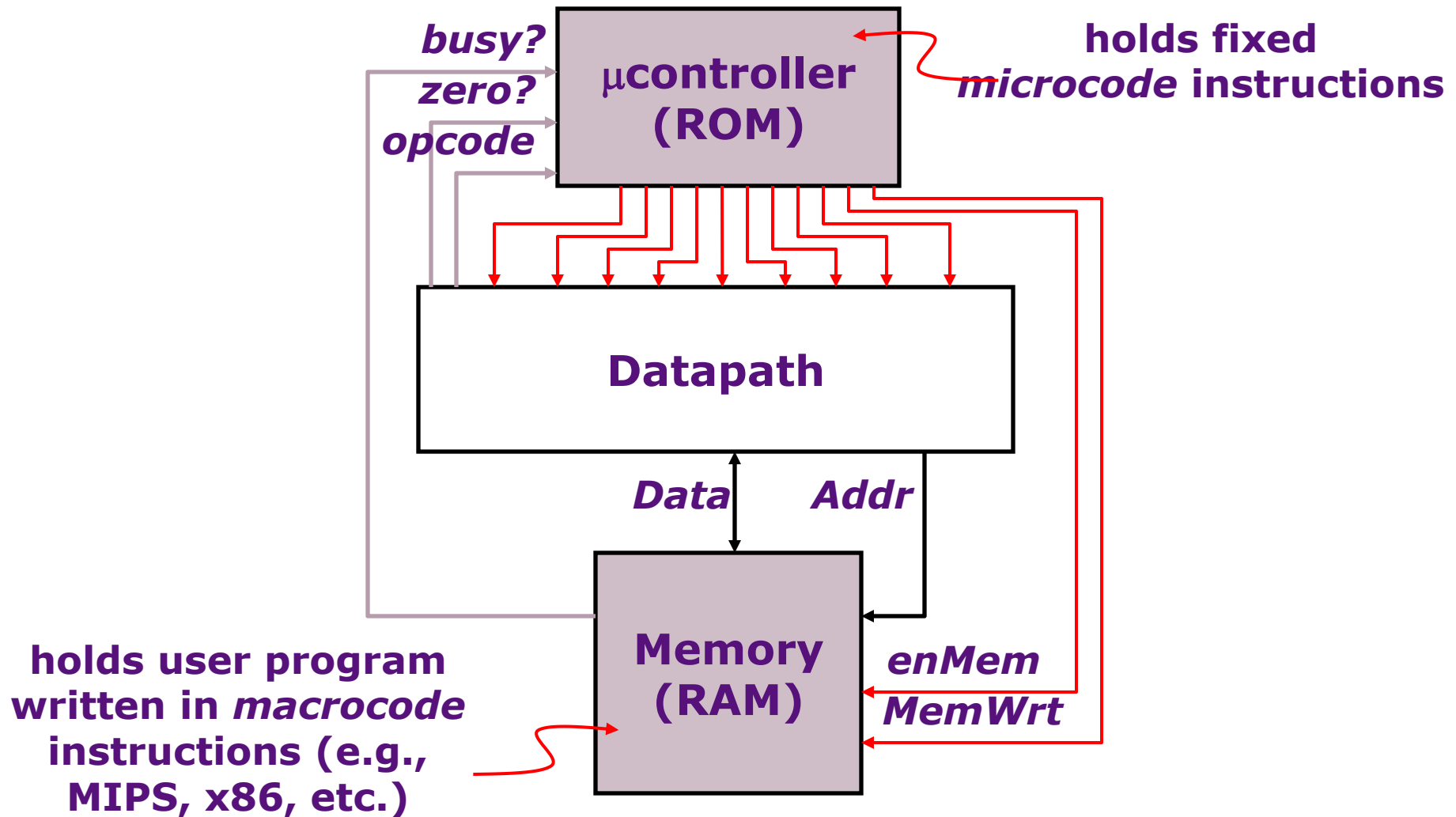
<http://www.csg.csail.mit.edu/6.823>

# Microcontrol Unit *Maurice Wilkes, 1954*

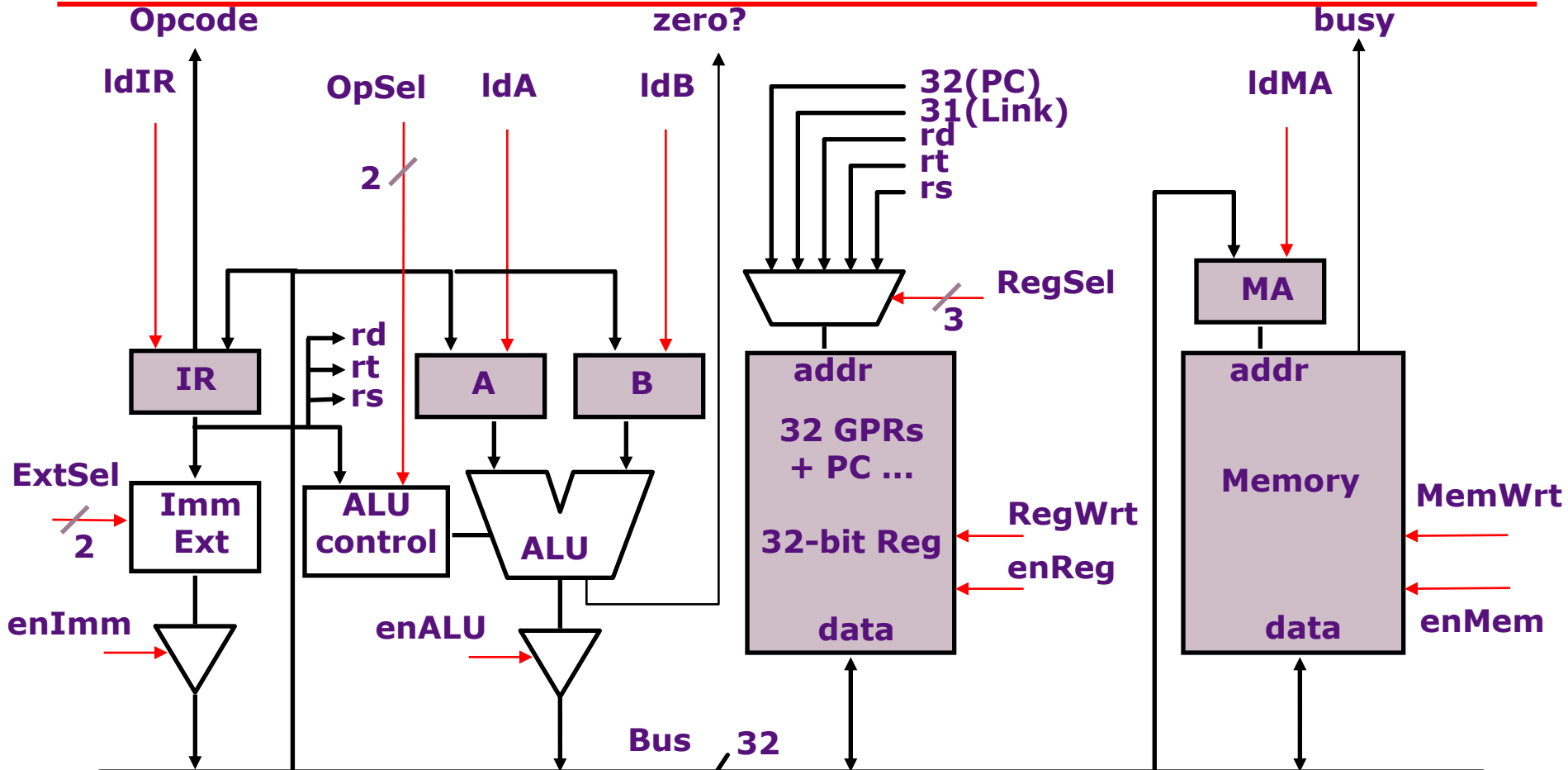
*Embed the control logic state table in a memory array*



# Microcoded Microarchitecture



# A Bus-based Datapath for MIPS

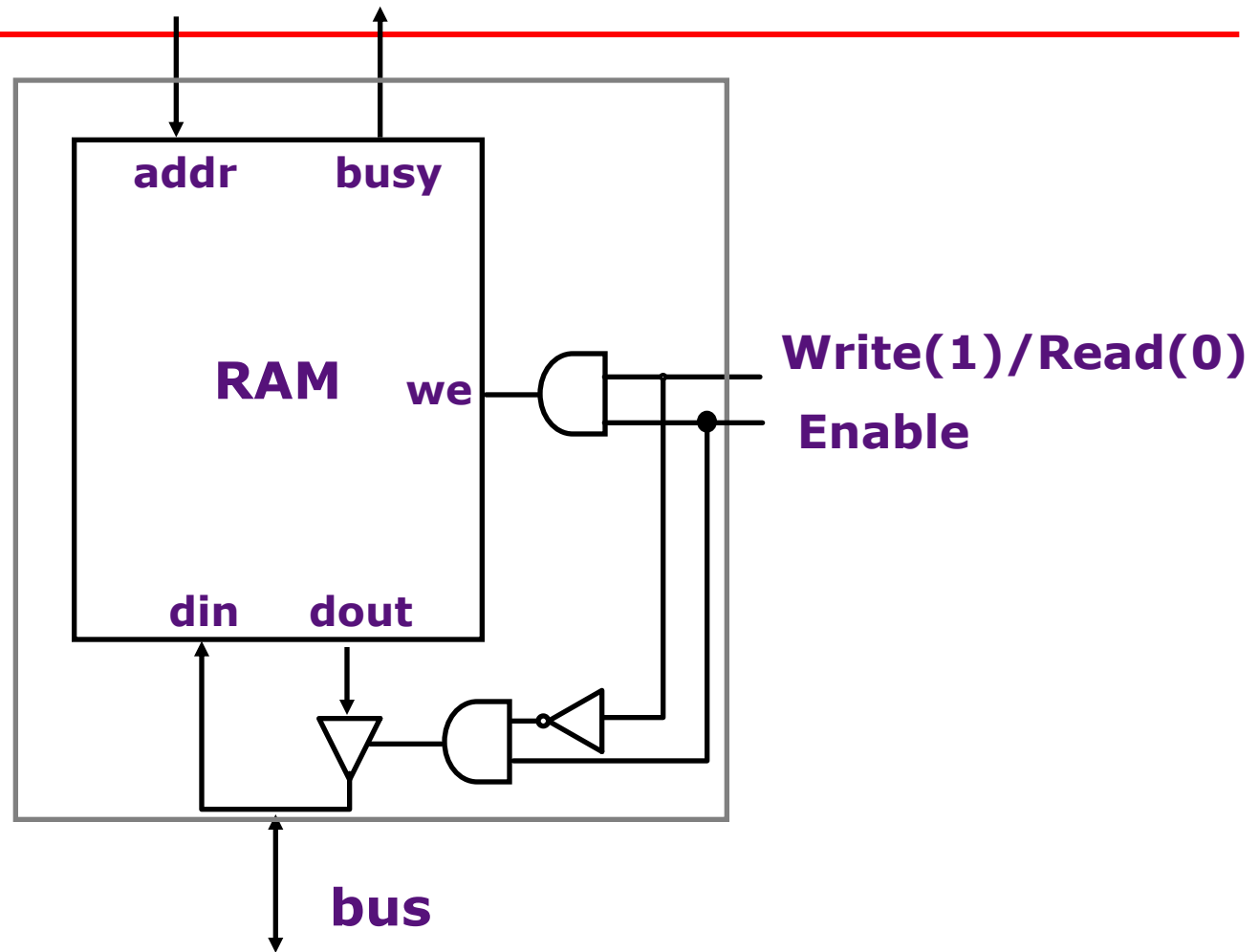


**Microinstruction: register to register transfer (17 control signals)**

**MA**  $\leftarrow$  PC    means    **RegSel = PC**;    **enReg=yes**;    **IdMA= yes**

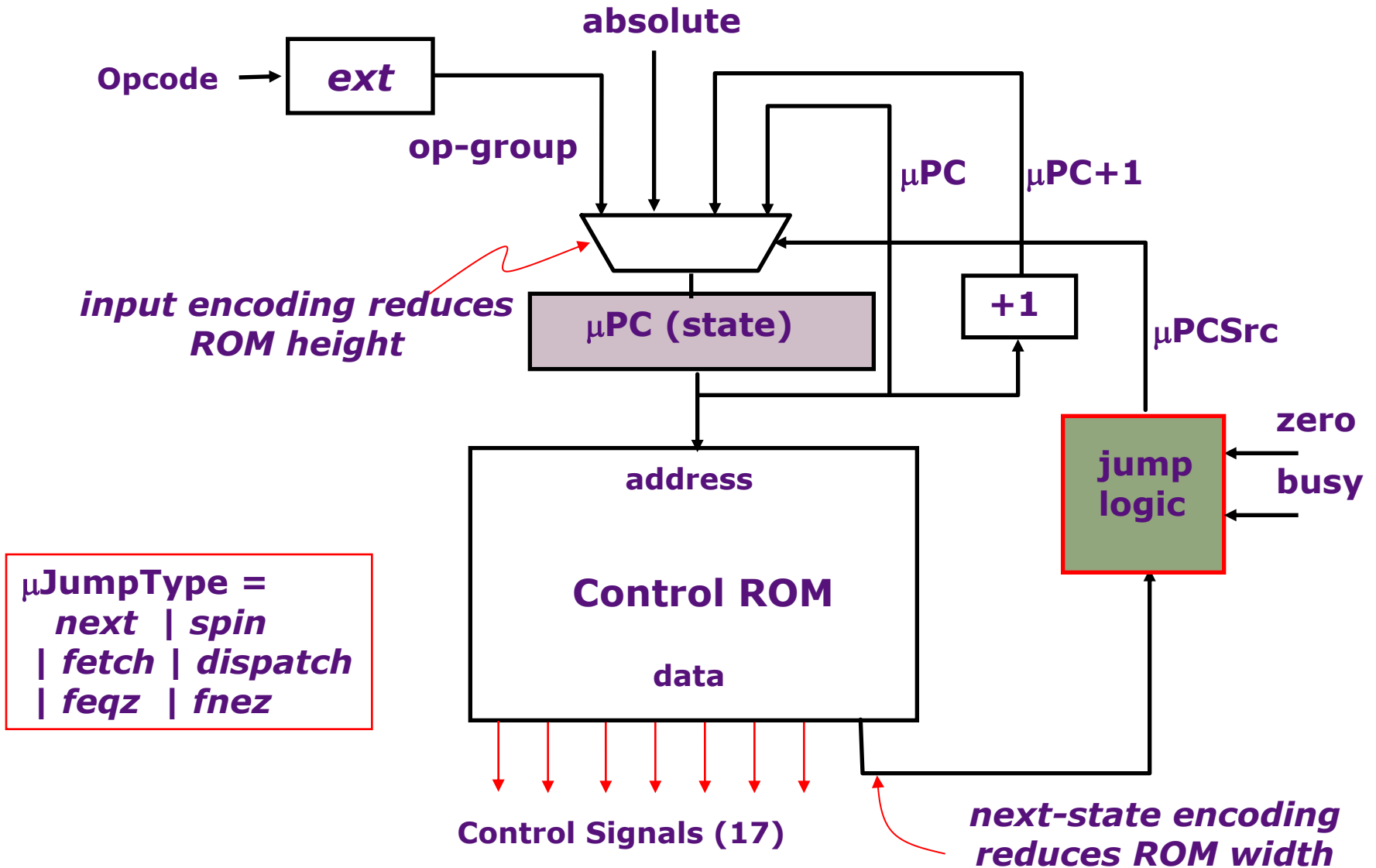
**B**  $\leftarrow$  Reg[rt] means    **RegSel = rt**;    **enReg=yes**;    **IdB = yes**

# Memory Module



**Assumption: Memory operates asynchronously and is slow as compared to Reg-to-Reg transfers**

# Microcode Controller



# Jump Logic

---

$\mu\text{PCSrc} = \text{Case } \mu\text{JumpTypes}$

**next**  $\Rightarrow$   $\mu\text{PC}+1$

**spin**  $\Rightarrow$  **if (busy) then  $\mu\text{PC}$  else  $\mu\text{PC}+1$**

**fetch**  $\Rightarrow$  **absolute**

**dispatch**  $\Rightarrow$  **op-group**

**feqz**  $\Rightarrow$  **if (zero) then absolute else  $\mu\text{PC}+1$**

**fnez**  $\Rightarrow$  **if (zero) then  $\mu\text{PC}+1$  else absolute**

# Instruction Execution

---

## Execution of a MIPS instruction involves

- 1. instruction fetch**
  - 2. decode and register fetch**
  - 3. ALU operation**
  - 4. memory operation (optional)**
  - 5. write back to register file (optional)**
- + the computation of the *next instruction* address**



# Instruction Fetch

---

State	Control points	next-state
fetch <sub>0</sub>	$MA \leftarrow PC$	<b>next</b>
fetch <sub>1</sub>	$IR \leftarrow \text{Memory}$	<b>spin</b>
fetch <sub>2</sub>	$A \leftarrow PC$	<b>next</b>
fetch <sub>3</sub>	$PC \leftarrow A + 4$	<b>dispatch</b>
...		
ALU <sub>0</sub>	$A \leftarrow \text{Reg}[rs]$	<b>next</b>
ALU <sub>1</sub>	$B \leftarrow \text{Reg}[rt]$	<b>next</b>
ALU <sub>2</sub>	$\text{Reg}[rd] \leftarrow \text{func}(A,B)$	<b>fetch</b>
ALUi <sub>0</sub>	$A \leftarrow \text{Reg}[rs]$	<b>next</b>
ALUi <sub>1</sub>	$B \leftarrow \text{sExt}_{16}(\text{Imm})$	<b>next</b>
ALUi <sub>2</sub>	$\text{Reg}[rd] \leftarrow \text{Op}(A,B)$	<b>fetch</b>

# Load & Store

---

State	Control points	next-state
<b>LW<sub>0</sub></b>	<b>A ← Reg[rs]</b>	<b>next</b>
<b>LW<sub>1</sub></b>	<b>B ← sExt<sub>16</sub>(Imm)</b>	<b>next</b>
<b>LW<sub>2</sub></b>	<b>MA ← A+B</b>	<b>next</b>
<b>LW<sub>3</sub></b>	<b>Reg[rt] ← Memory</b>	<b>spin</b>
<b>LW<sub>4</sub></b>		<b>fetch</b>
<b>SW<sub>0</sub></b>	<b>A ← Reg[rs]</b>	<b>next</b>
<b>SW<sub>1</sub></b>	<b>B ← sExt<sub>16</sub>(Imm)</b>	<b>next</b>
<b>SW<sub>2</sub></b>	<b>MA ← A+B</b>	<b>next</b>
<b>SW<sub>3</sub></b>	<b>Memory ← Reg[rt]</b>	<b>spin</b>
<b>SW<sub>4</sub></b>		<b>fetch</b>

# Branches

---

State	Control points	next-state
<b>BEQZ<sub>0</sub></b>	<b>A ← Reg[rs]</b>	<b>next</b>
<b>BEQZ<sub>1</sub></b>		<b>fnez</b>
<b>BEQZ<sub>2</sub></b>	<b>A ← PC</b>	<b>next</b>
<b>BEQZ<sub>3</sub></b>	<b>B ← sExt<sub>16</sub>(Imm&lt;&lt;2)</b>	<b>next</b>
<b>BEQZ<sub>4</sub></b>	<b>PC ← A+B</b>	<b>fetch</b>
<b>BNEZ<sub>0</sub></b>	<b>A ← Reg[rs]</b>	<b>next</b>
<b>BNEZ<sub>1</sub></b>		<b>feqz</b>
<b>BNEZ<sub>2</sub></b>	<b>A ← PC</b>	<b>next</b>
<b>BNEZ<sub>3</sub></b>	<b>B ← sExt<sub>16</sub>(Imm&lt;&lt;2)</b>	<b>next</b>
<b>BNEZ<sub>4</sub></b>	<b>PC ← A+B</b>	<b>fetch</b>

# Jumps

---

State	Control points	next-state
$J_0$	$A \leftarrow PC$	next
$J_1$	$B \leftarrow IR$	next
$J_2$	$PC \leftarrow \text{JumpTarg}(A,B)$	fetch
$JR_0$	$A \leftarrow \text{Reg}[rs]$	next
$JR_1$	$PC \leftarrow A$	fetch
$JAL_0$	$A \leftarrow PC$	next
$JAL_1$	$\text{Reg}[31] \leftarrow A$	next
$JAL_2$	$B \leftarrow IR$	next
$JAL_3$	$PC \leftarrow \text{JumpTarg}(A,B)$	fetch
$JALR_0$	$A \leftarrow PC$	next
$JALR_1$	$B \leftarrow \text{Reg}[rs]$	next
$JALR_2$	$\text{Reg}[31] \leftarrow A$	next
$JALR_3$	$PC \leftarrow B$	fetch

# VAX 11-780 Microcode

```

      ; P1WFUD,1 [600,1205]      MICRO2 1F(12)      26-May-81 14:58:1      VAX11/780 Microcode : PCS 01, FPLA 0D, WCS122      Page 771
      ; CALL2 ,MIC [600,1205]      Procedure call      : CALLG, CALLS

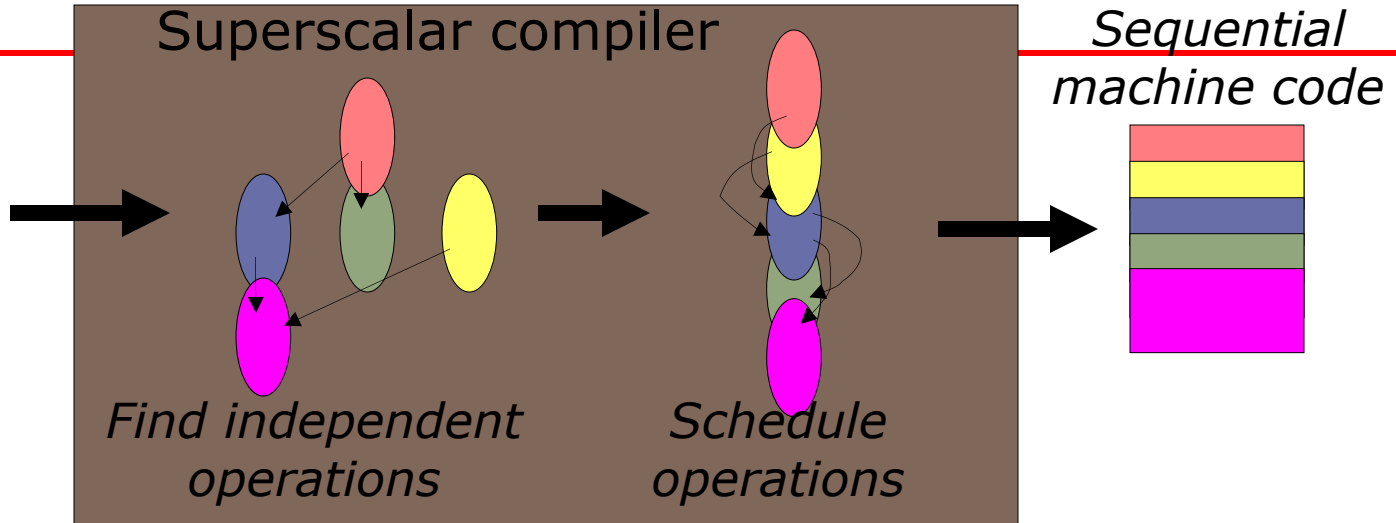
      ;29744 ;HERE FOR CALLG OR CALLS, AFTER PROBING THE EXTENT OF THE STACK
      ;29745
      ;29746 =0 ;-----;CALL SITE FOR MPUSH
      ;29747 CALL,7: D_Q,AND,RC[T2], ;STRIP MASK TO BITS 11-0
      6557K 0 U 11F4, 0811,2035,0180,F910,0000,0CD8 ;29748 CALL,J/MPUSH ;PUSH REGISTERS
      ;29749
      ;29750 ;-----;RETURN FROM MPUSH
      ;29751 CACHE_D[LONG], ;PUSH PC
      6557K 7763K U 11F5, 0000,003C,0180,3270,0000,134A ;29752 LAB_R[SP] ; BY SP
      ;29753
      ;29754 ;-----;
      6856K 0 U 134A, 0018,0000,0180,FAF0,0200,134C ;29755 CALL,8: R[SP]&VA_LA=K[,8] ;UPDATE SP FOR PUSH OF PC &
      ;29756
      ;29757 ;-----;
      6856K 0 U 134C, 0800,003C,0180,FA68,0000,11F8 ;29758 D_R[FP] ;READY TO PUSH FRAME POINTER
      ;29759
      ;29760 =0 ;-----;CALL SITE FOR PSHSP
      ;29761 CACHE_D[LONG], ;STORE FP,
      ;29762 LAB_R[SP], ; GET SP AGAIN
      ;29763 SC_K[.FFF0], ;-16 TO SC
      6856K 21M U 11F8, 0000,003D,6D80,3270,0084,6CD9 ;29764 CALL,J/PSHSP
      ;29765
      ;29766 ;-----;
      ;29767 D_R[AP], ;READY TO PUSH AP
      6856K 0 U 11F9, 0800,003C,3DF0,2E60,0000,134D ;29768 Q_ID[PSL] ; AND GET PSW FOR COMBINATIO
      ;29769
      ;29770 ;-----;
      ;29771 CACHE_D[LONG], ;STORE OLD AP
      ;29772 Q_Q,ANDNOT,K[.1F], ;CLEAR PSW<T,N,Z,V,C>
      6856K 21M U 134D, 0019,2024,8DC0,3270,0000,134E ;29773 LAB_R[SP] ;GET SP INTO LATCHES AGAIN
      ;29774
      ;29775 ;-----;
      6856K 0 U 134E, 2010,0038,0180,F909,4200,1350 ;29776 PC&VA_RC[T1], FLUSH,IB ; LOAD NEW PC AND CLEAR OUT
      ;29777
      ;29778 ;-----;
      ;29779 D_DAL,SC, ;PSW TO D<31:16>
      ;29780 Q_RC[T2], ;RECOVER MASK
      ;29781 SC=SC+K[,3], ;PUT -13 IN SC
      6856K 0 U 1350, 0D10,0038,0DC0,6114,0084,9351 ;29782 LOAD,IB, PC_PC+1 ;START FETCHING SUBROUTINE I
      ;29783
      ;29784 ;-----;
      ;29785 D_DAL,SC, ;MASK AND PSW IN D<31:03>
      ;29786 Q_PC[T4], ;GET LOW BITS OF OLD SP TO Q<1:0>
      6856K 0 U 1351, 0D10,0038,F5C0,F920,0084,9352 ;29787 SC=SC+K[,A] ;PUT -3 IN SC
      ;29788
  
```

# Sequential ISA Bottleneck

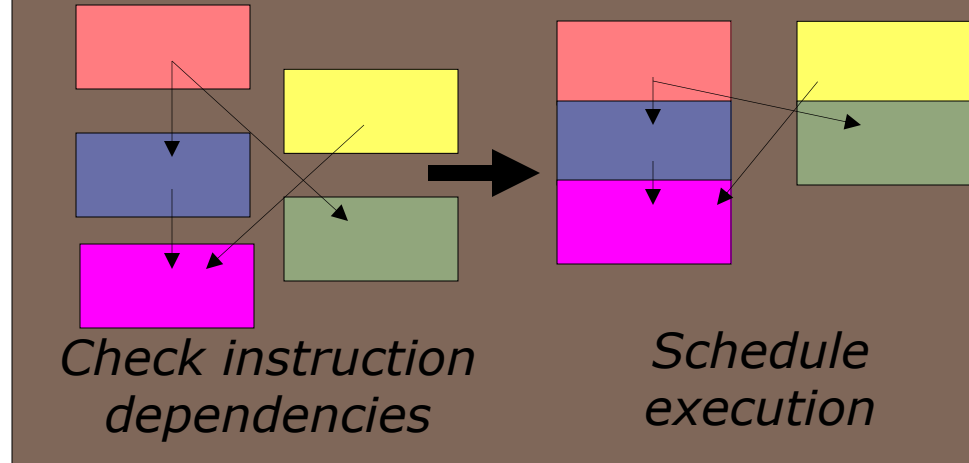
*Sequential  
source code*

```
a = foo(b);
for (i=0, i<
```

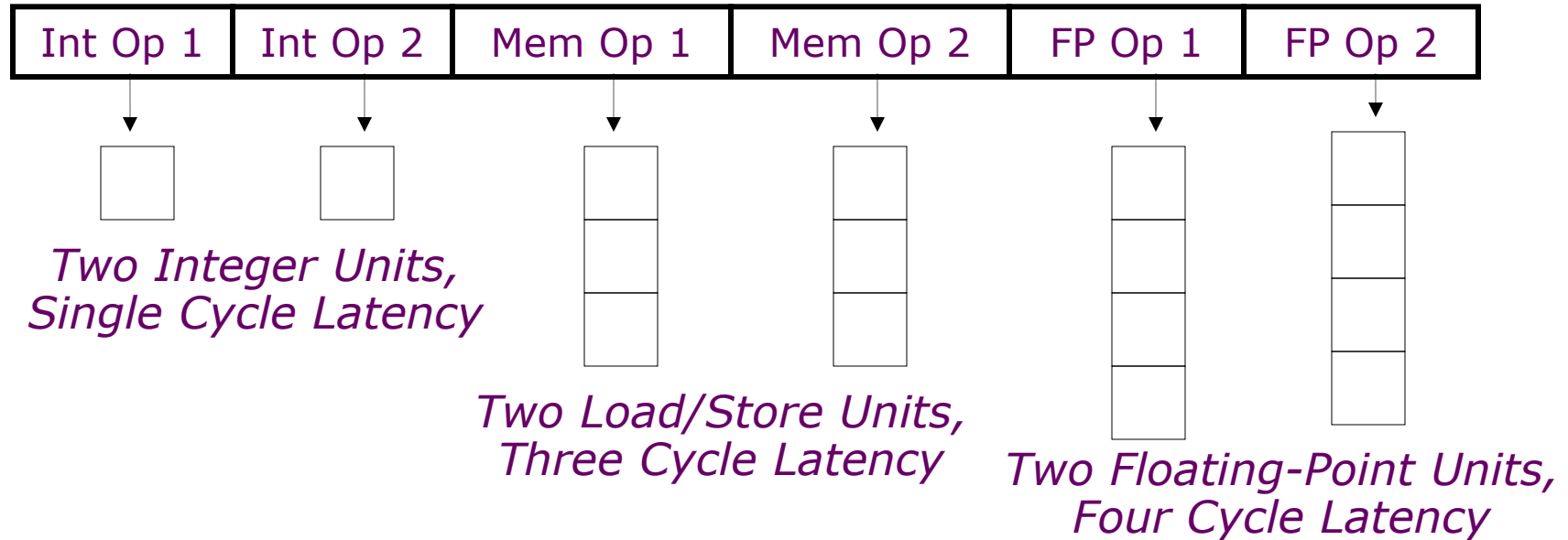
Superscalar compiler



Superscalar processor



# VLIW: Very Long Instruction Word



- Multiple operations packed into one instruction
- Each operation slot is for a fixed function
- Constant operation latencies are specified

# VLIW Design Principles

---

## The architecture:

- Allows operation parallelism within an instruction
  - No x-operation RAW check
- Provides deterministic latency for all operations
  - Latency measured in 'instructions'
  - No data use allowed before specified latency with no data interlocks

## The compiler:

- Schedules (reorders) to maximize parallel execution
- Guarantees intra-instruction parallelism
- Schedules to avoid data hazards (no interlocks)
  - Typically separates operations with explicit NOPs



# Early VLIW Machines

---

- **FPS AP120B (1976)**
  - scientific attached array processor
  - first commercial wide instruction machine
  - hand-coded vector math libraries using software pipelining and loop unrolling
- **Multiflow Trace (1987)**
  - commercialization of ideas from Fisher's Yale group including "trace scheduling"
  - available in configurations with 7, 14, or 28 operations/instruction
  - 28 operations packed into a 1024-bit instruction word
- **Cydrome Cydra-5 (1987)**
  - 7 operations encoded in 256-bit instruction word
  - rotating register file

# Loop Execution

```
for (i=0; i<N; i++)
  B[i] = A[i] + C;
```

*Compile*

```
loop: ld f1, 0(r1)
      add r1, 8
      fadd f2, f0, f1
      sd f2, 0(r2)
      add r2, 8
      bne r1, r3, loop
```

loop:

*Schedule*

	Int1	Int 2	M1	M2	FP+	FPx
add r1			ld			
					fadd	
add r2	bne		sd			

How many FP ops/cycle?

**1 fadd / 8 cycles = 0.125**

# Loop Unrolling

---

```
for (i=0; i<N; i++)  
    B[i] = A[i] + C;
```

Unroll inner loop to  
perform 4 iterations  
at once

```
for (i=0; i<N; i+=4)  
{  
    B[i]    = A[i] + C;  
    B[i+1] = A[i+1] + C;  
    B[i+2] = A[i+2] + C;  
    B[i+3] = A[i+3] + C;  
}
```

Is this code correct?

No, need to handle values of N that are not multiples of unrolling factor with final cleanup loop

# Scheduling Loop Unrolled Code

*Unroll 4 ways*

```

loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      sd f8, 24(r2)
      add r2, 32
      bne r1, r3, loop
  
```

loop:

*Schedule* →

Int1	Int 2	M1	M2	FP+	FPx
		ld f1			
		ld f2			
		ld f3			
add r1		ld f4		fadd f5	
				fadd f6	
				fadd f7	
				fadd f8	
		sd f5			
		sd f6			
		sd f7			
add r2	bne	sd f8			

How many FLOPS/cycle?

**4 fadds / 11 cycles = 0.36**

# Software Pipelining

*Unroll 4 ways first*

```

loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      add r2, 32
      sd f8, -8(r2)
      bne r1, r3, loop
  
```

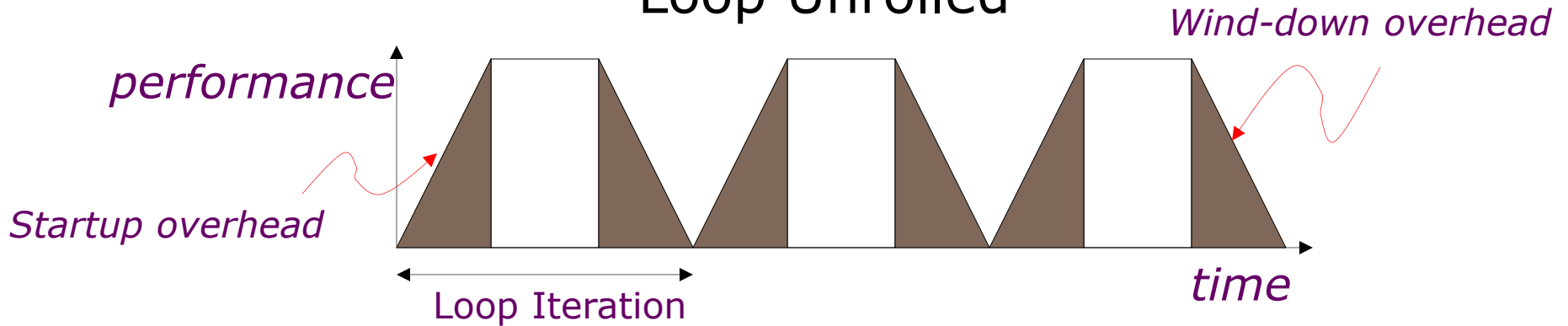
	Int1	Int 2	M1	M2	FP+	FPx
prolog			ld f1			
			ld f2			
			ld f3			
	add r1		ld f4			
			ld f1		fadd f5	
			ld f2		fadd f6	
			ld f3		fadd f7	
	add r1		ld f4		fadd f8	
iterate			ld f1	sd f5	fadd f5	
			ld f2	sd f6	fadd f6	
		add r2	ld f3	sd f7	fadd f7	
	add r1	bne	ld f4	sd f8	fadd f8	
epilog				sd f5	fadd f5	
				sd f6	fadd f6	
		add r2		sd f7	fadd f7	
		bne		sd f8	fadd f8	
				sd f5		

How many FLOPS/cycle?

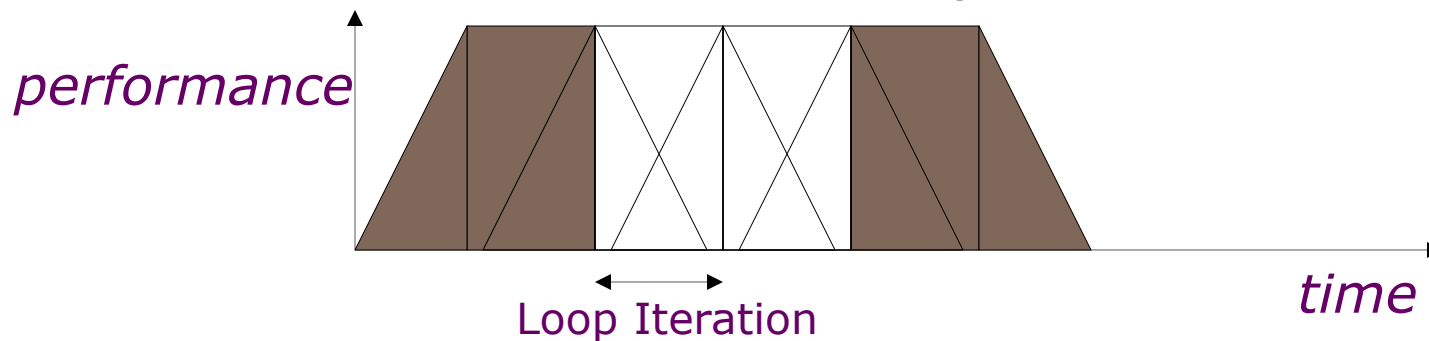
$$4 \text{ fadds} / 4 \text{ cycles} = 1$$

# Software Pipelining vs. Loop Unrolling

## Loop Unrolled



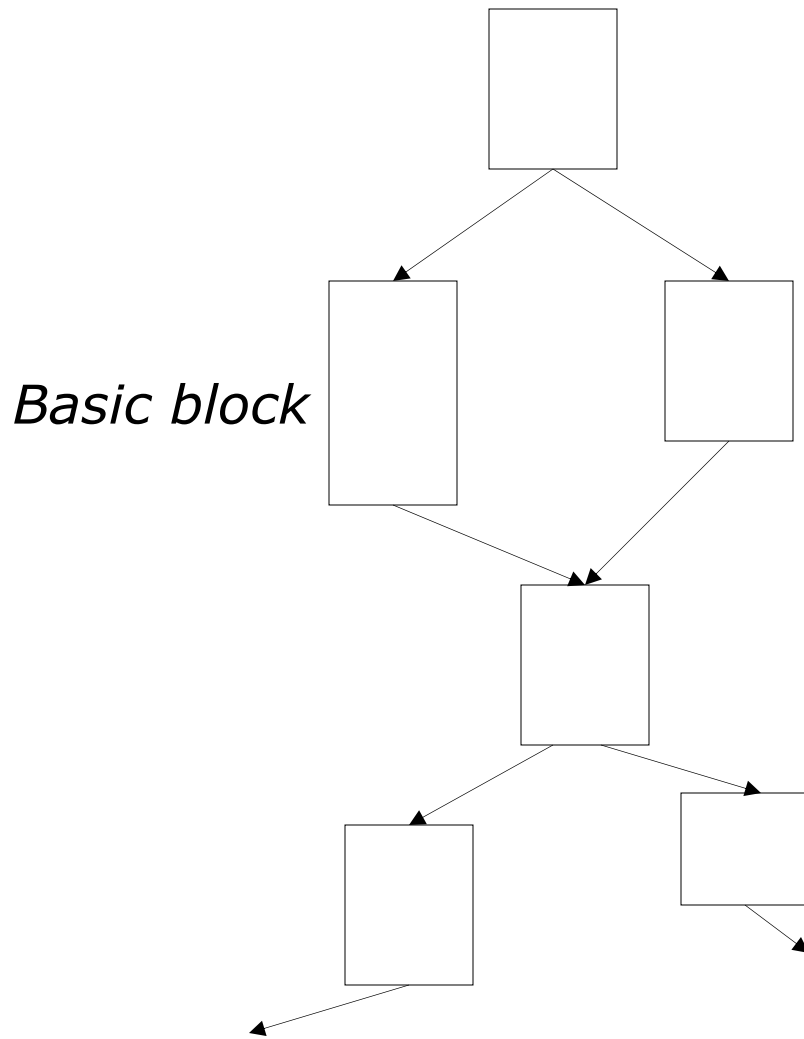
## Software Pipelined



*Software pipelining pays startup/wind-down costs only once per loop, not once per iteration*

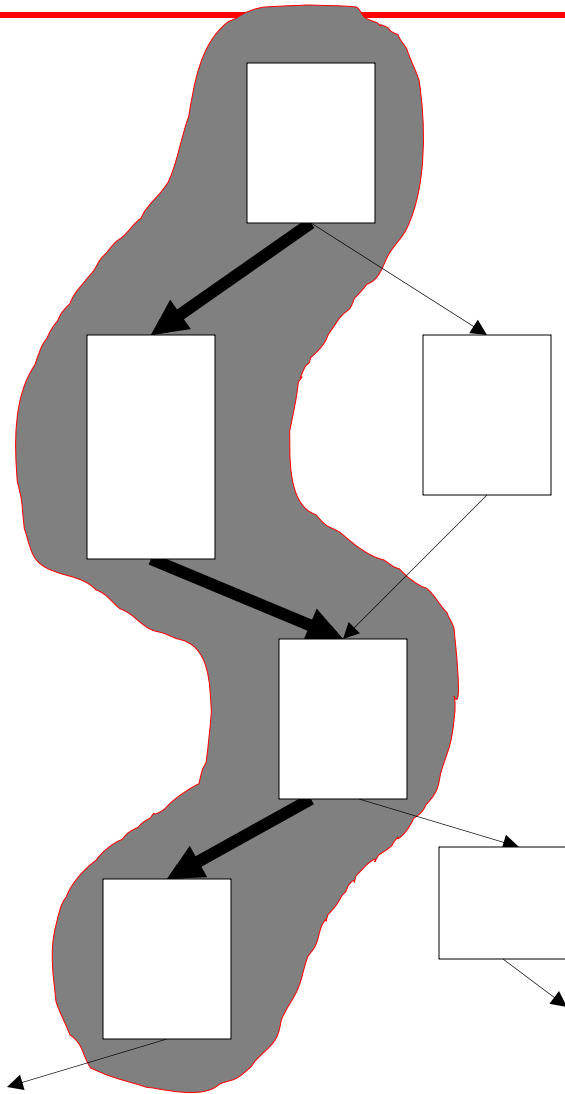
# What if there are no loops?

---



- Branches limit basic block size in control-flow intensive irregular code
- Difficult to find ILP in individual basic blocks

# Trace Scheduling [Fisher, Ellis]



- Pick string of basic blocks, a *trace*, that represents most frequent branch path
- Schedule whole "trace" at once
- Add fixup code to cope with branches jumping out of trace

How do we know which trace to pick?

Use profiling feedback or compiler heuristics to find common branch paths



# Problems with “Classic” VLIW

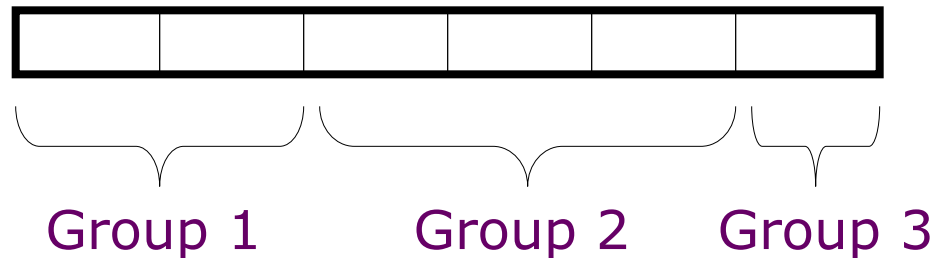
---

- Knowing branch probabilities
  - Profiling requires a significant extra step in build process
- Scheduling for statically unpredictable branches
  - optimal schedule varies with branch path
- Object code size
  - instruction padding wastes instruction memory/cache
  - loop unrolling/software pipelining replicates code
- Scheduling memory operations
  - caches and/or memory bank conflicts impose statically unpredictable variability
  - uncertainty about addresses limit code reordering
- Object-code compatibility
  - have to recompile all code for every machine, even for two machines in same generation

# VLIW Instruction Encoding

---

- Schemes to reduce effect of unused fields
  - Compressed format in memory, expand on I-cache refill
    - used in Multiflow Trace
    - introduces instruction addressing challenge
  - Provide a single-op VLIW instruction
    - Cydra-5 UniOp instructions
  - Mark parallel groups
    - used in TMS320C6x DSPs, Intel IA-64



# Cydra-5: Memory Latency Register (MLR)

---

Problem: Loads have variable latency

Solution: Let software choose desired memory latency

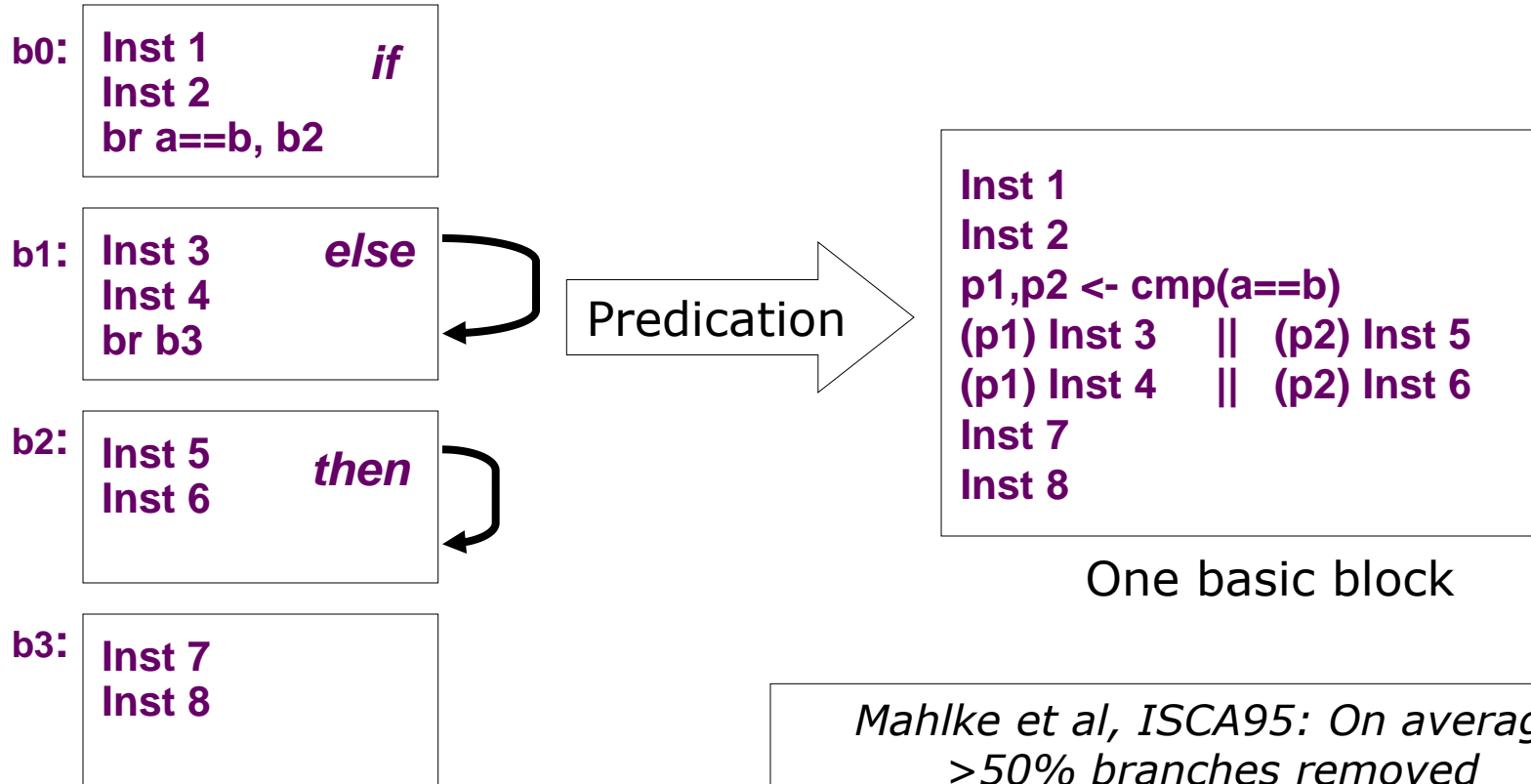
- Compiler schedules code for maximum load-use distance
- Software sets MLR to latency that matches code schedule
- Hardware ensures that loads take exactly MLR cycles to return values into processor pipeline
  - Hardware buffers loads that return early
  - Hardware stalls processor if loads return late

# IA-64 Predicated Execution

Problem: Mispredicted branches limit ILP

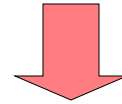
Solution: Eliminate hard to predict branches with predicated execution

- Almost all IA-64 instructions can be executed conditionally under predicate
- Instruction becomes NOP if predicate register false

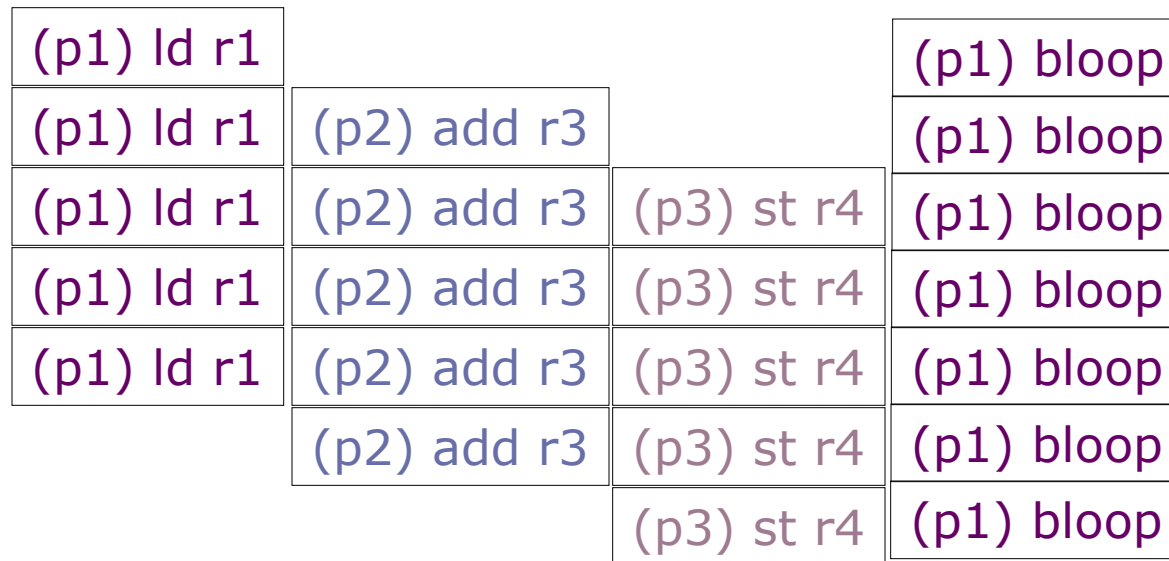


# Predicate Software Pipeline Stages

## Single VLIW Instruction

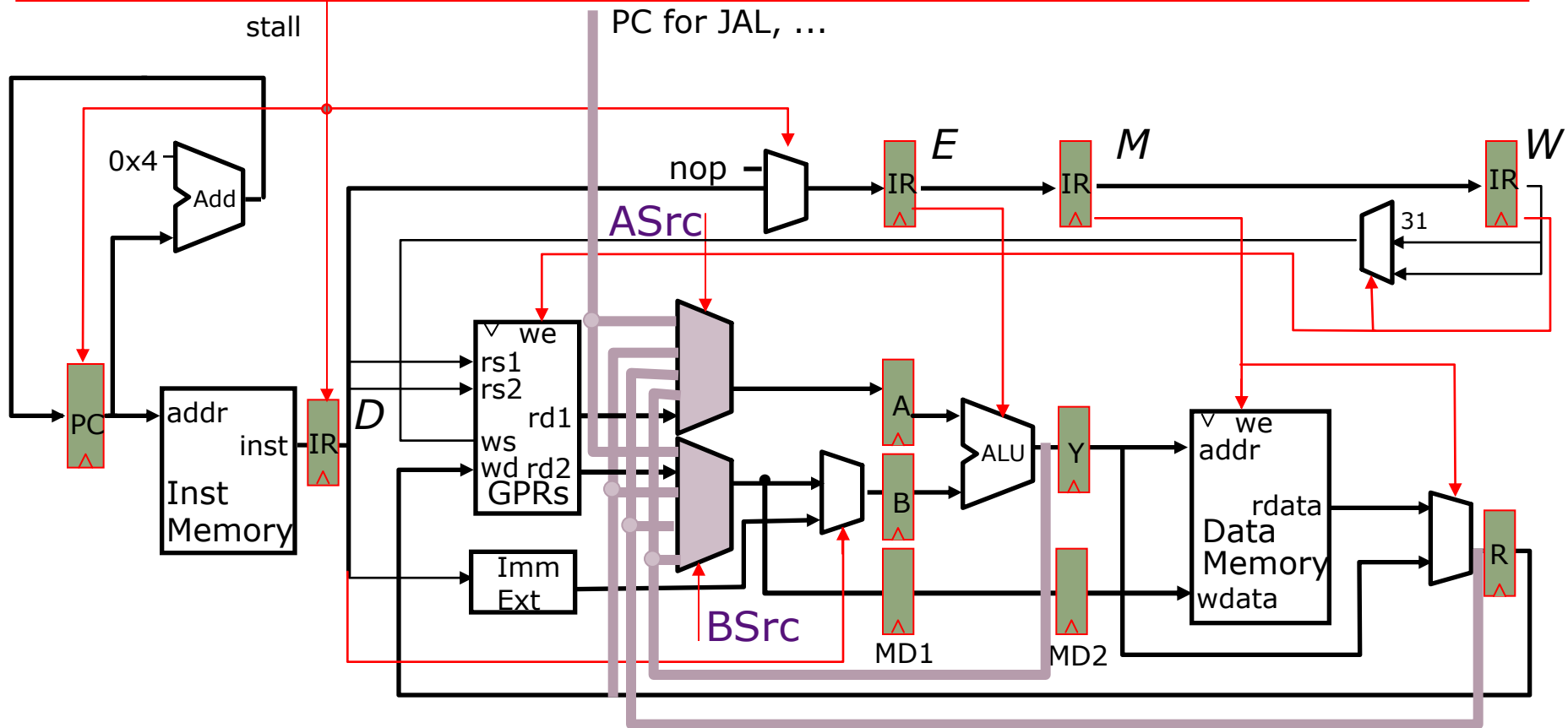


## Dynamic Execution



Software pipeline stages turned on by rotating predicate registers → Much denser encoding of loops

# Fully Bypassed Datapath



Where does predication fit in?

# IA-64 Speculative Execution

Problem: Branches restrict compiler code motion

Solution: Speculative operations that don't cause exceptions

Inst 1  
Inst 2  
br a==b, b2

Load r1  
Use r1  
Inst 3

*Can't move load above branch  
because might cause spurious  
exception*

Load.s r1  
Inst 1  
Inst 2  
br a==b, b2

Chk.s r1  
Use r1  
Inst 3

*Speculative load  
never causes  
exception, but sets  
"poison" bit on  
destination register*

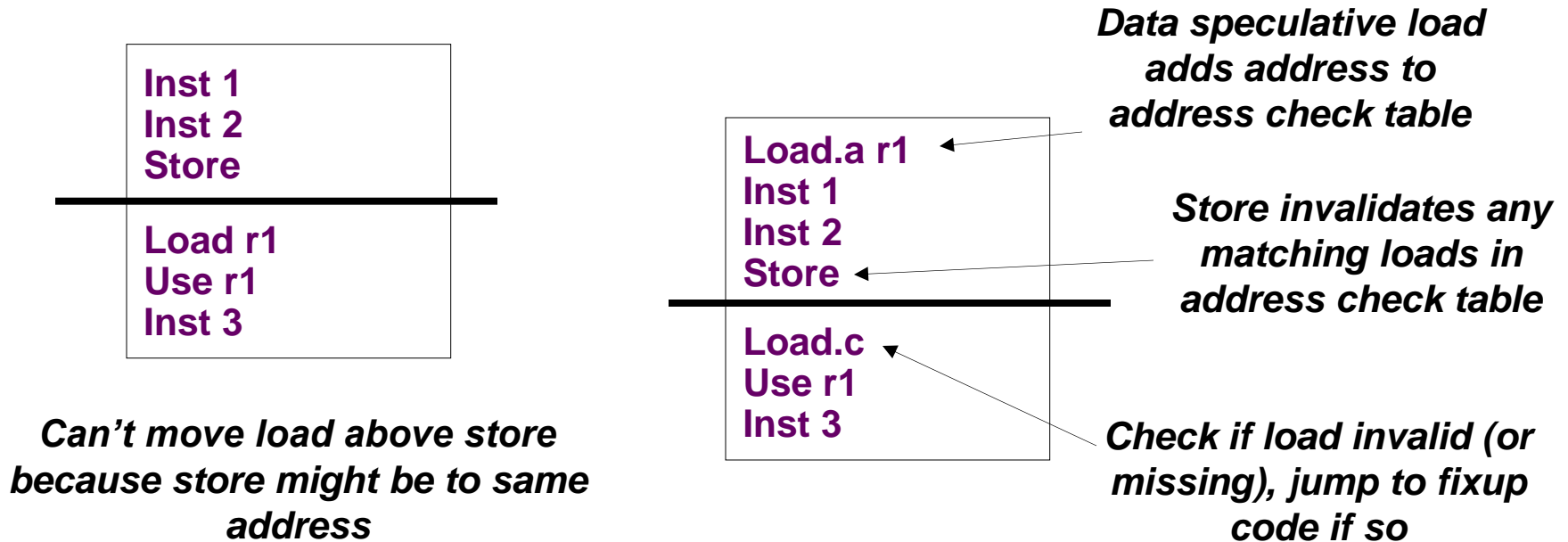
*Check for exception in  
original home block  
jumps to fixup code if  
exception detected*

Particularly useful for scheduling long latency loads early

# IA-64 Data Speculation

Problem: Possible memory hazards limit code scheduling

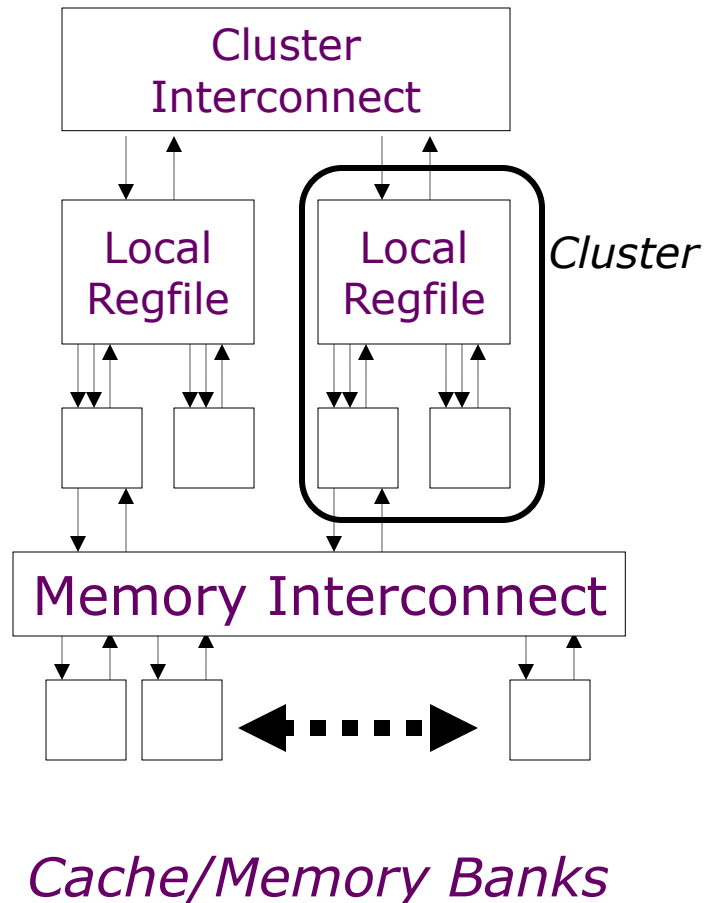
Solution: Instruction-based speculation with hardware monitor to check for pointer hazards



Requires associative hardware in address check table



# Clustered VLIW



- Divide machine into clusters of local register files and local functional units
- Lower bandwidth/higher latency interconnect between clusters
- Software responsible for mapping computations to minimize communication overhead
- Common in commercial embedded processors, examples include TI C6x series DSPs, and HP Lx processor
- Exists in some superscalar processors, .e.g., Alpha 21264

# Limits of Static Scheduling

---

- Unpredictable branches
- Unpredictable memory behavior (cache misses and dependencies)
- Code size explosion
- Compiler complexity

## Question:

How applicable are the VLIW-inspired techniques to traditional RISC/CISC processor architectures?



*Thank you !*

<http://www.csg.csail.mit.edu/6.823>