Microcoded and VLIW Processors

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

Hardwired vs Microcoded Processors

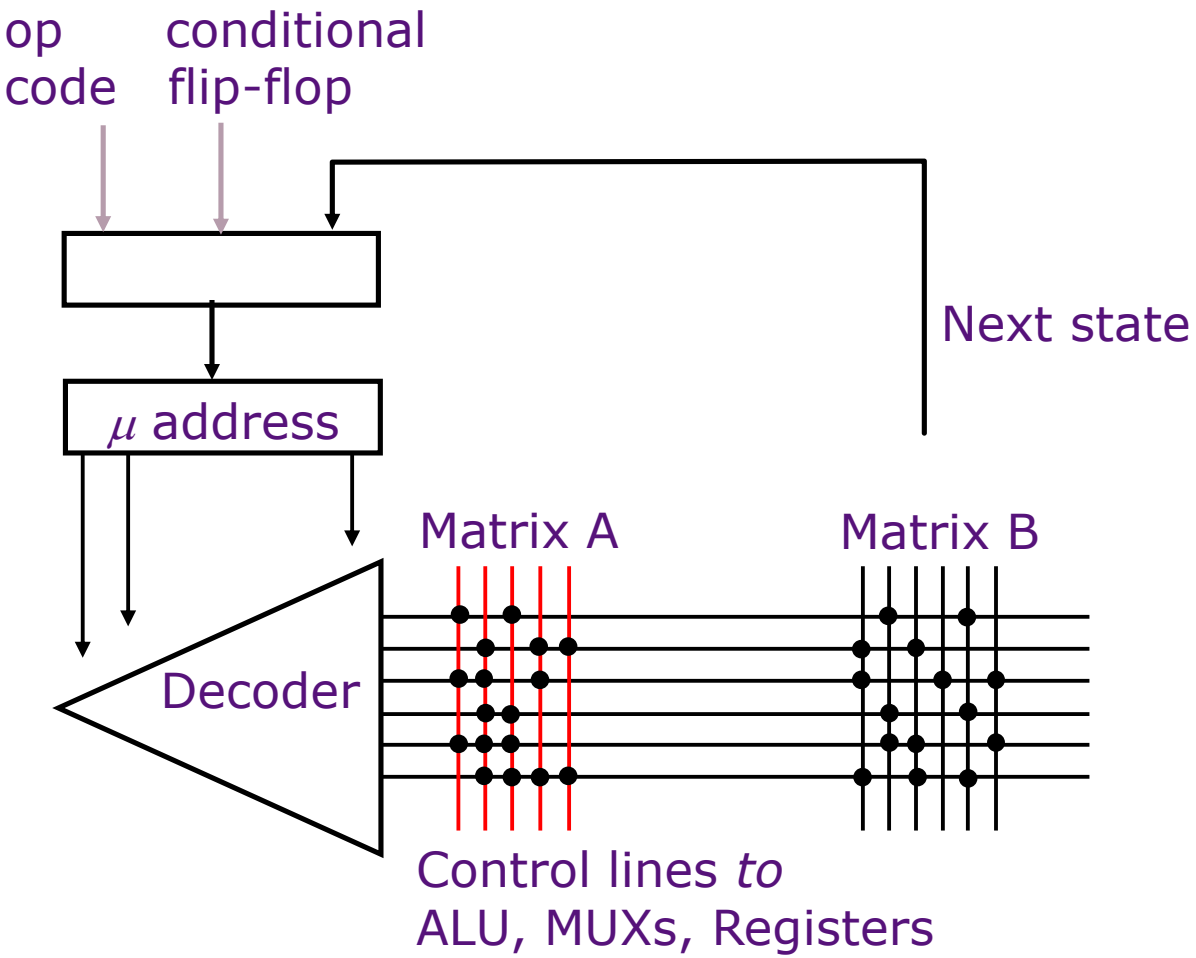
- All processors we have seen so far are hardwired: The microarchitecture directly implements all the instructions in the ISA

- Microcoded processors add a layer of interpretation: Each ISA instruction is executed as a sequence of simpler *microinstructions*
  - *Simpler implementation*
  - *Lower performance than hardwired (CPI > 1)*

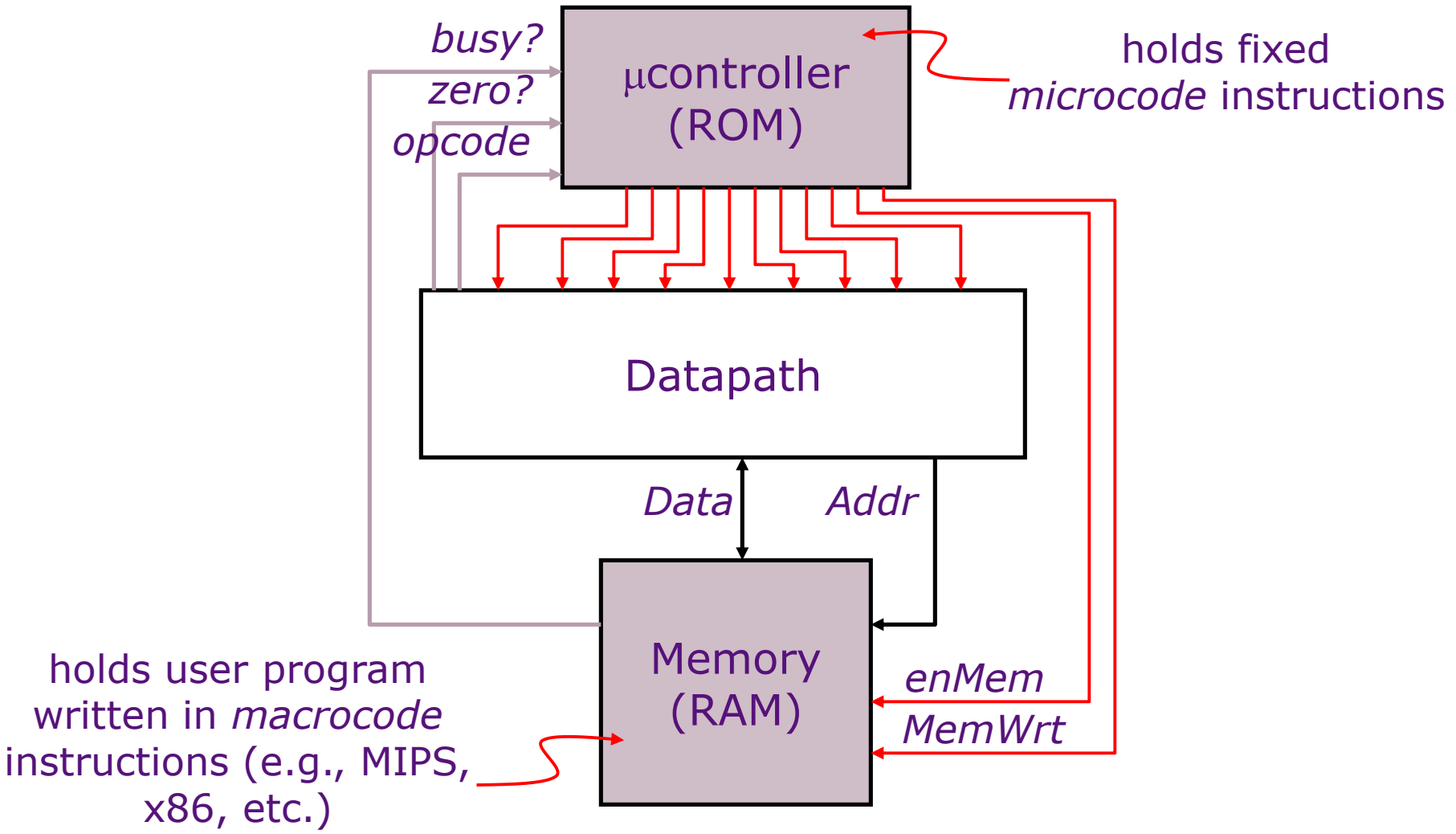
- Microcoding common until the 80s, still in use today (e.g., complex x86 instructions are decoded into multiple "micro-ops")

Microcontrol Unit
[Maurice Wilkes, 1954]

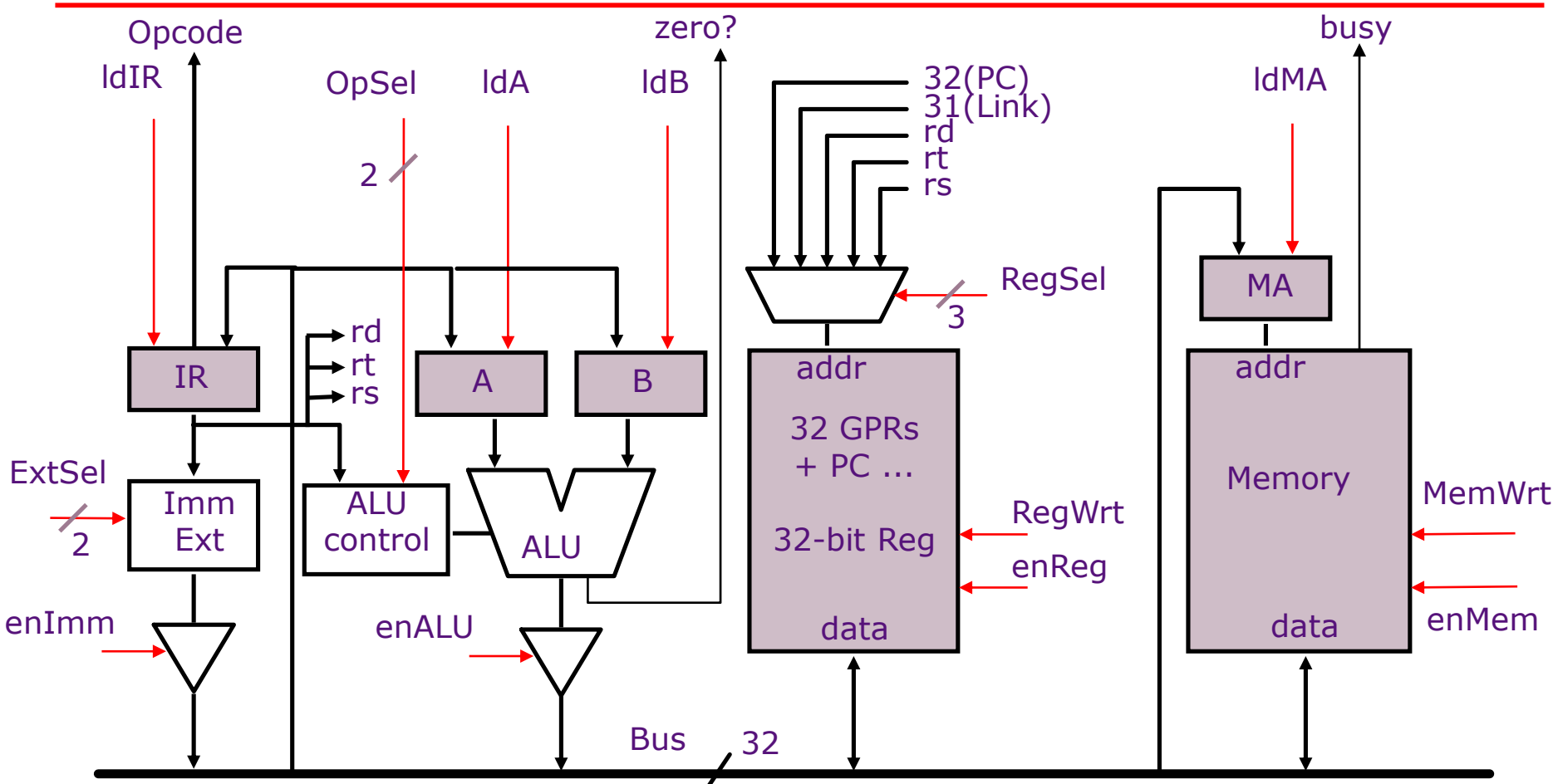
Embed the control logic state table in a read-only memory array



Microcoded Microarchitecture



A Bus-based Datapath for MIPS

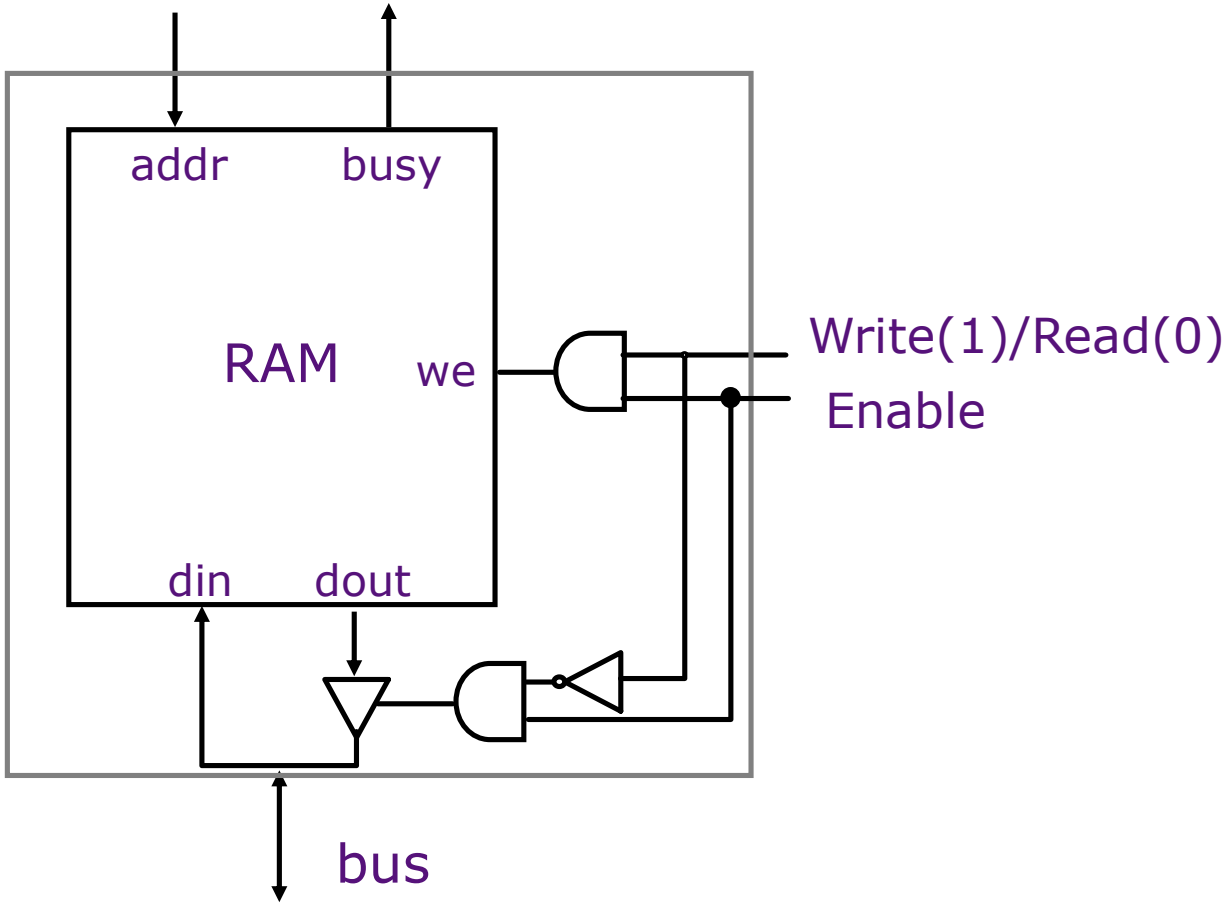


Microinstruction: register to register transfer (17 control signals)

MA ← PC *means* RegSel = PC; enReg=yes; ldMA= yes

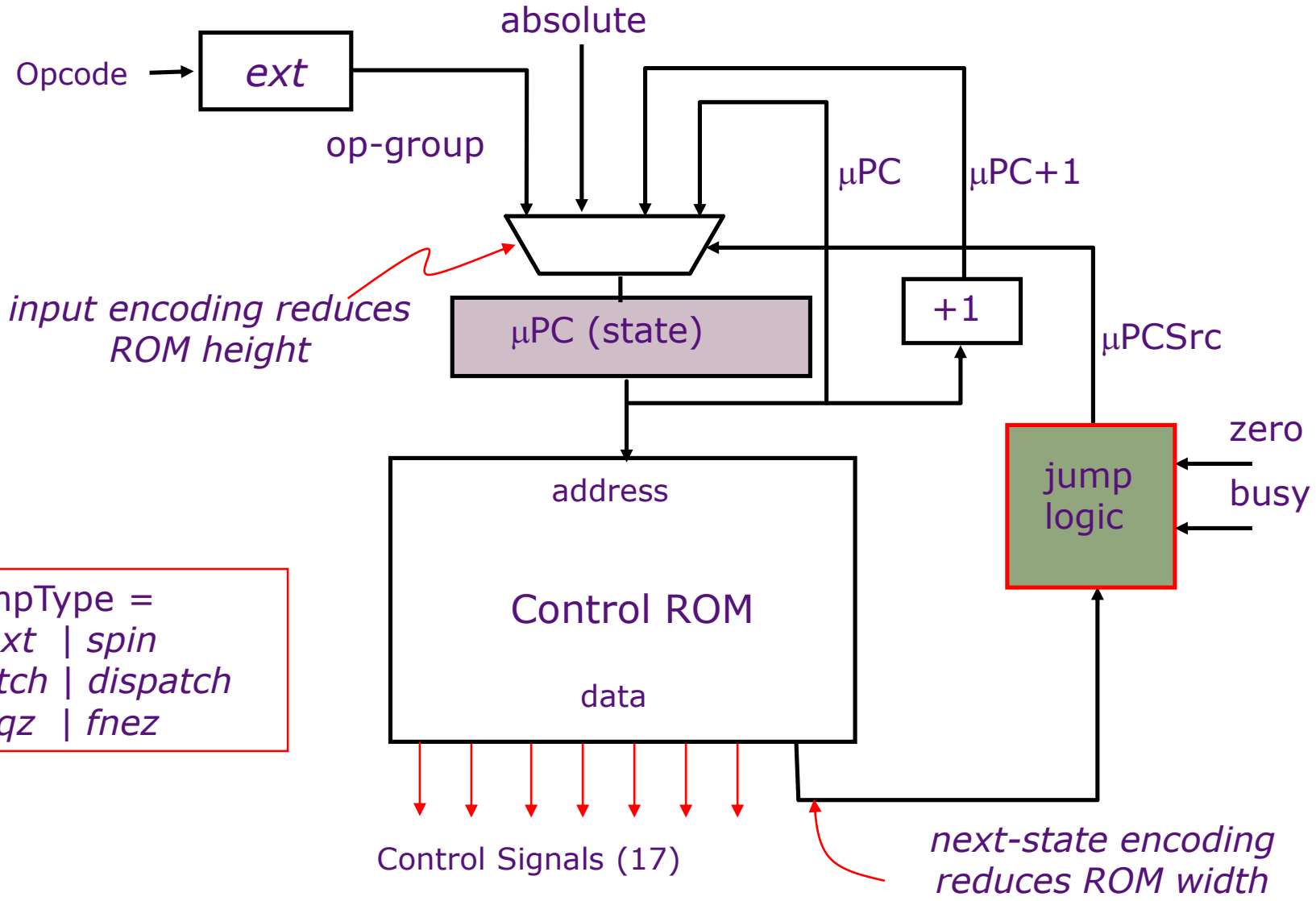
B ← Reg[rt] *means* RegSel = rt; enReg=yes; ldB = yes

Memory Module



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

Microcode Controller



Jump Logic

μPCSrc = *Case* μJumpTypes

next	⇒	μPC+1
spin	⇒	if (busy) then μPC else μPC+1
fetch	⇒	absolute
dispatch	⇒	op-group
feqz	⇒	if (zero) then absolute else μPC+1
fnez	⇒	if (zero) then μPC+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
$fetch_0$	MA ← PC	next
$fetch_1$	IR ← Memory	spin
$fetch_2$	A ← PC	next
$fetch_3$	PC ← A + 4	dispatch
...		
ALU_0	A ← Reg[rs]	next
ALU_1	B ← Reg[rt]	next
ALU_2	Reg[rd]←func(A,B)	fetch
$ALUi_0$	A ← Reg[rs]	next
$ALUi_1$	B ← $sExt_{16}$(Imm)	next
$ALUi_2$	Reg[rd]← Op(A,B)	fetch

Load & Store

State	Control points	next-state
LW_0	A $\leftarrow$ Reg[rs]	next
LW_1	B $\leftarrow$ $sExt_{16}$(Imm)	next
LW_2	MA $\leftarrow$ A+B	next
LW_3	Reg[rt] $\leftarrow$ Memory	spin
LW_4		fetch
SW_0	A $\leftarrow$ Reg[rs]	next
SW_1	B $\leftarrow$ $sExt_{16}$(Imm)	next
SW_2	MA $\leftarrow$ A+B	next
SW_3	Memory $\leftarrow$ Reg[rt]	spin
SW_4		fetch

Branches

State	Control points	next-state
$BEQZ_0$	A ← Reg[rs]	next
$BEQZ_1$		fnez
$BEQZ_2$	A ← PC	next
$BEQZ_3$	B ← $sExt_{16}$(Imm<<2)	next
$BEQZ_4$	PC ← A+B	fetch
$BNEZ_0$	A ← Reg[rs]	next
$BNEZ_1$		feqz
$BNEZ_2$	A ← PC	next
$BNEZ_3$	B ← $sExt_{16}$(Imm<<2)	next
$BNEZ_4$	PC ← A+B	fetch

Jumps

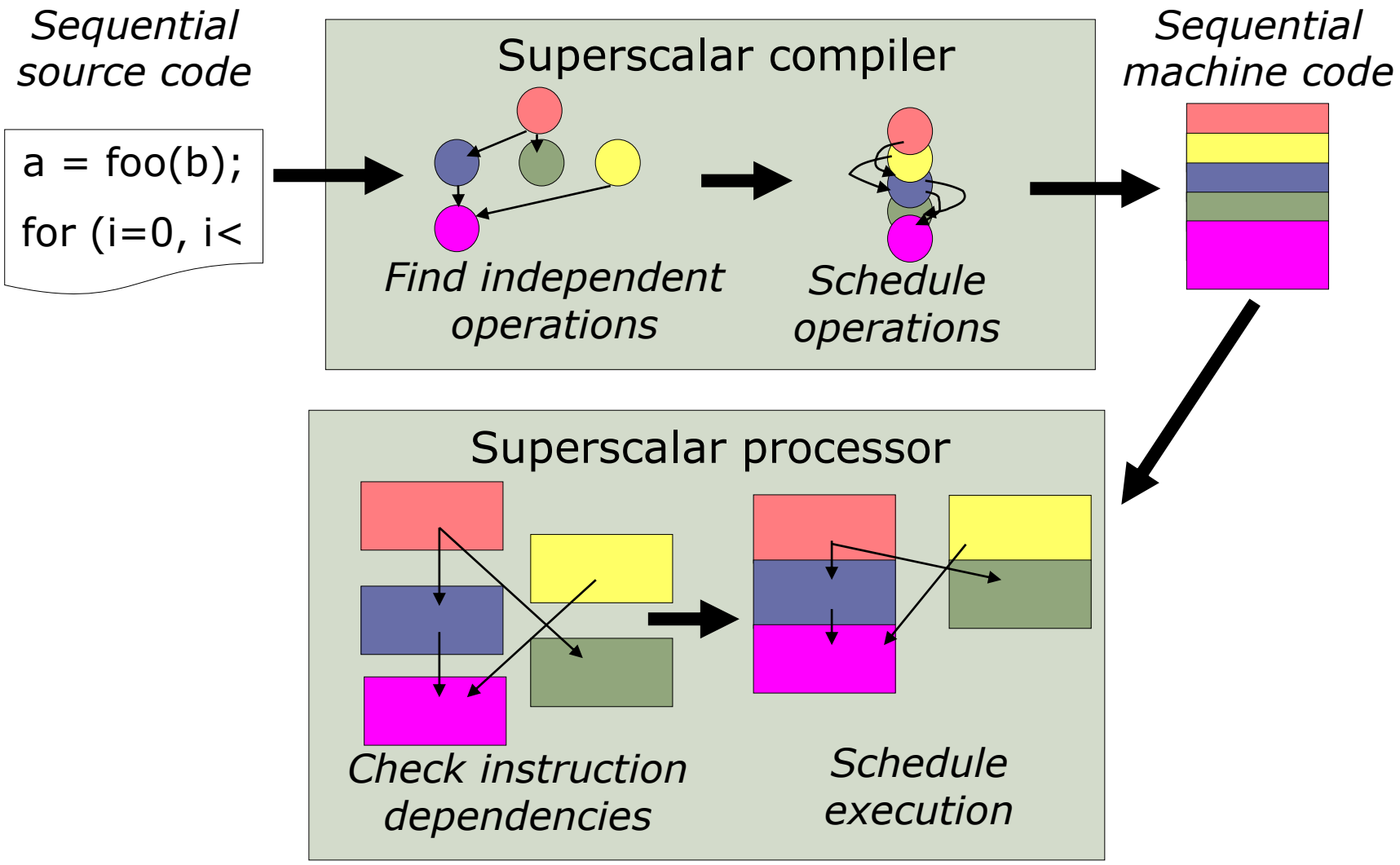
State	Control points	next-state
J_0	A ← PC	next
J_1	B ← IR	next
J_2	PC ← JumpTarg(A,B)	fetch
JR_0	A ← Reg[rs]	next
JR_1	PC ← A	fetch
JAL_0	A ← PC	next
JAL_1	Reg[31] ← A	next
JAL_2	B ← IR	next
JAL_3	PC ← JumpTarg(A,B)	fetch
$JALR_0$	A ← PC	next
$JALR_1$	B ← Reg[rs]	next
$JALR_2$	Reg[31] ← A	next
$JALR_3$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

VAX 11-780 Microcode (1978)

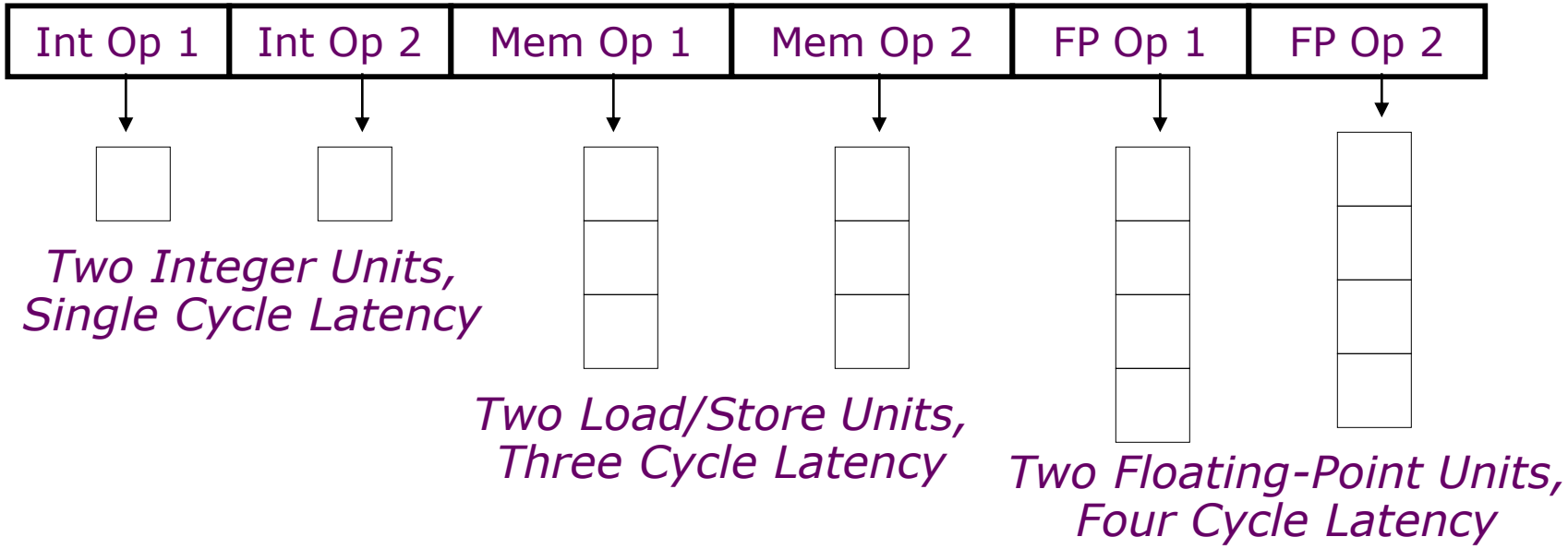
```
; P1WFUD.M  [600,1205]      MICRO2  1F(12)     26-May-81  14:58:1      VAX11/780 Microcode : PCS 01, FPLA 0D, WCS122      Page  771
; CALL2 .MIC [600,1205]     Procedure call          : CALLG, CALLS

                                                ;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
```

Very Long Instruction Word (VLIW) Processors

Sequential ISA Bottleneck



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 cross-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

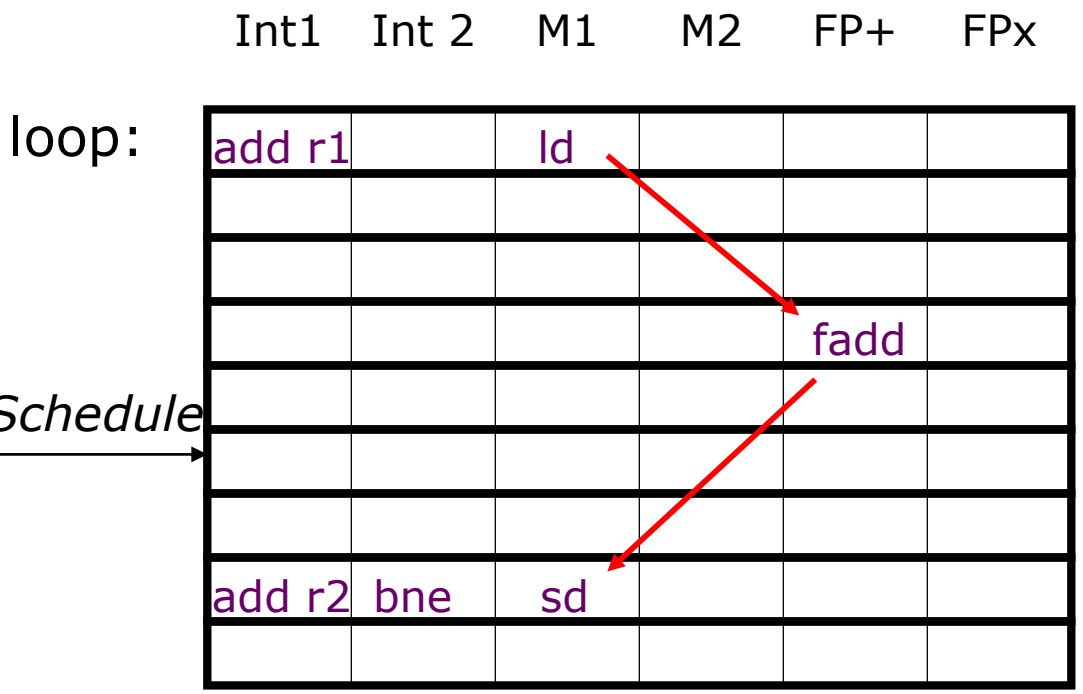
Loop Execution

```
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
```

Schedule



How many FP ops/cycle?

1 fadd / 8 cycles = 0.125

Loop Unrolling

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

```
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;
}
```

Unroll inner loop to perform 4 iterations at once

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
```

Schedule →

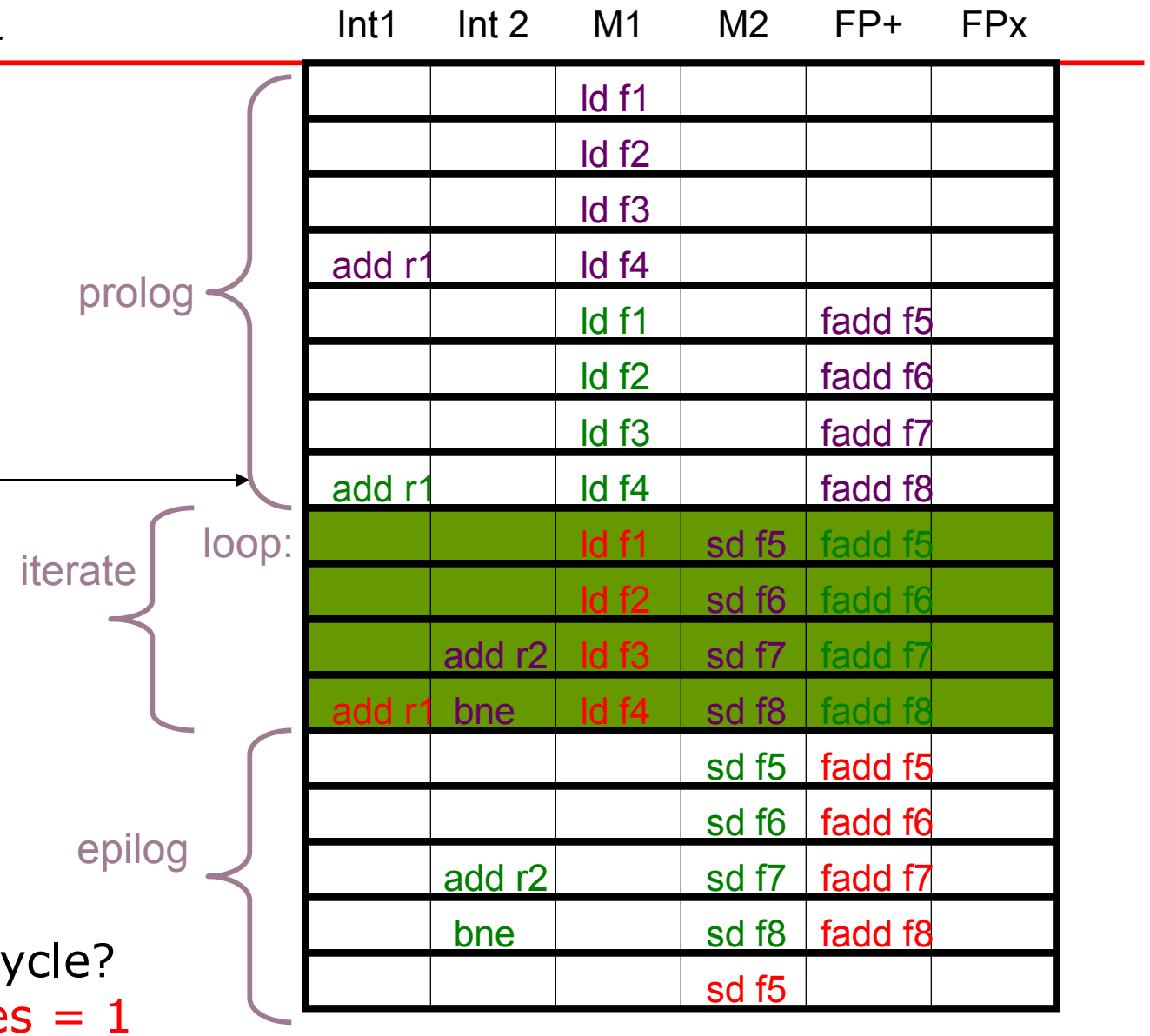
	Int1	Int 2	M1	M2	FP+	FPx
loop:			ld f1			
			ld f2			
			ld f3			
	add r1		ld f4		fadd f5	
					fadd f6	
					fadd f7	
					fadd f8	
			sd f5			
			sd f6			
			sd f7			
	add r2	bne	sd f8			

How many FLOPS/cycle? 4 fadds / 11 cycles = 0.36

Software Pipelining

Unroll 4 ways first

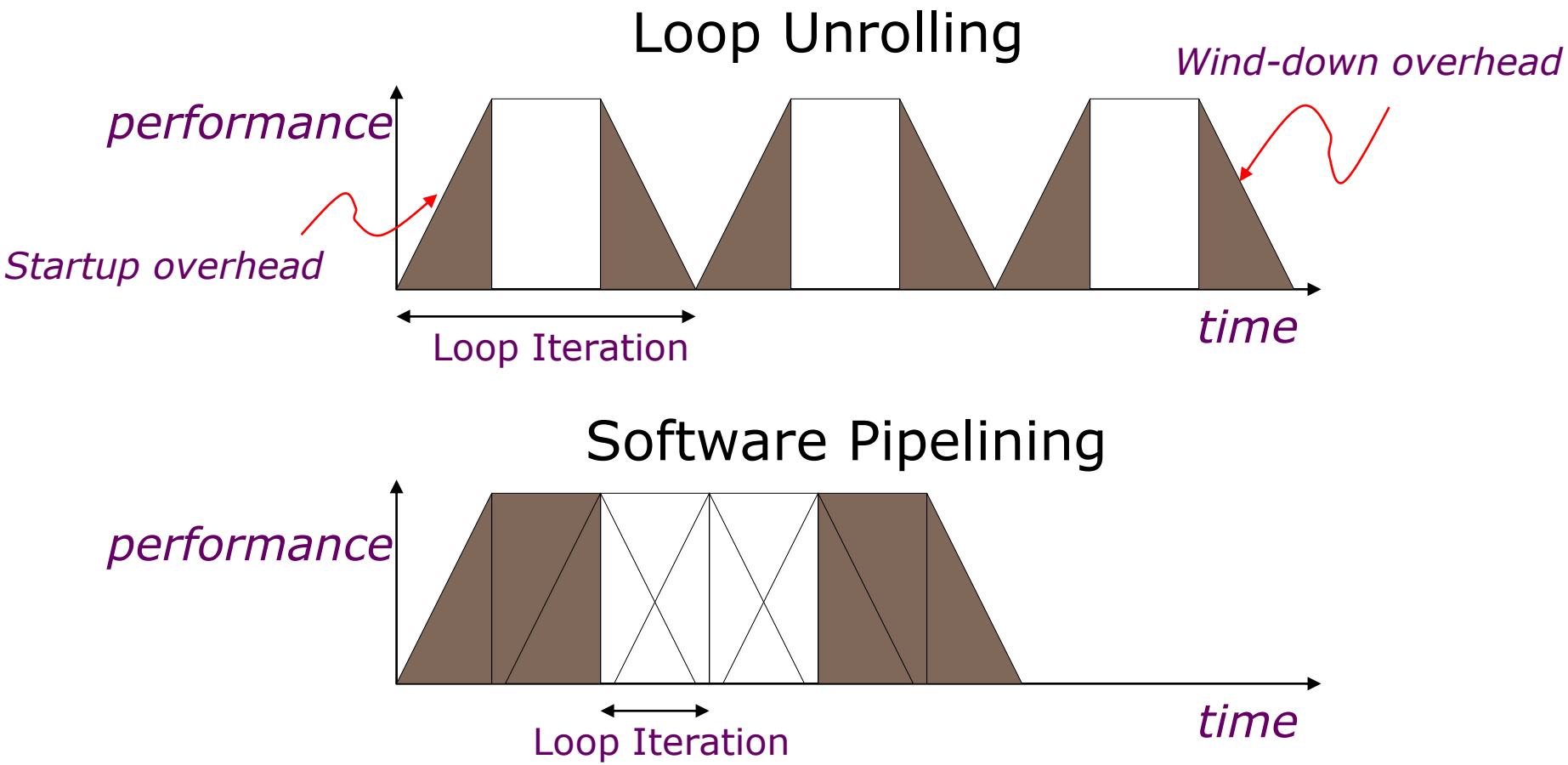
```
loop:  ld f1, 0(r1)
       ld f2, 8(r1)
       ld f3, 16(r1)
       ld f4, 24(r1)
       add r1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       sd f5, 0(r2)
       sd f6, 8(r2)
       sd f7, 16(r2)
       add r2, 32
       sd f8, -8(r2)
    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



How many FLOPS/cycle?

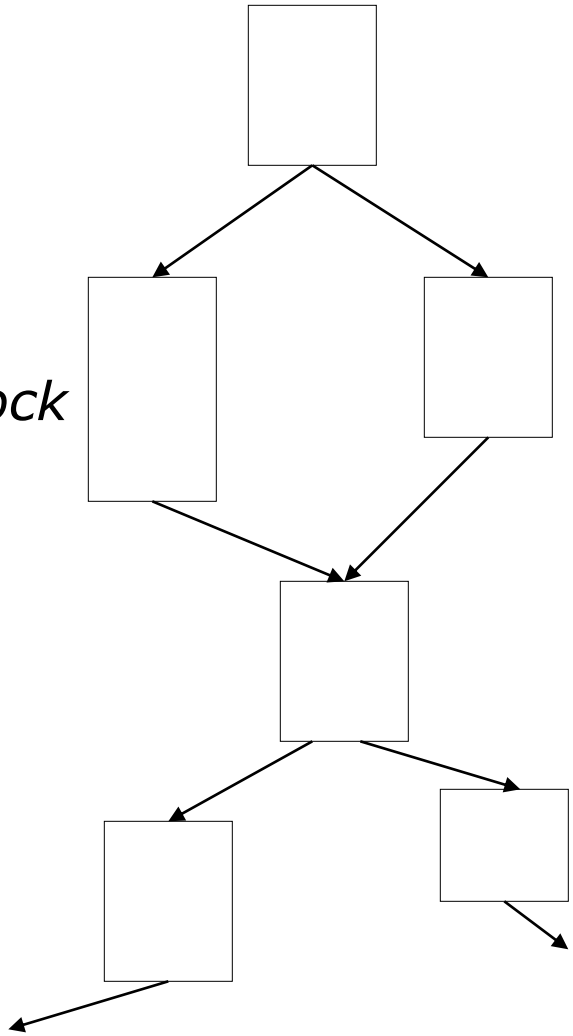
4 fadds / 4 cycles = 1

Software Pipelining vs. Unrolling



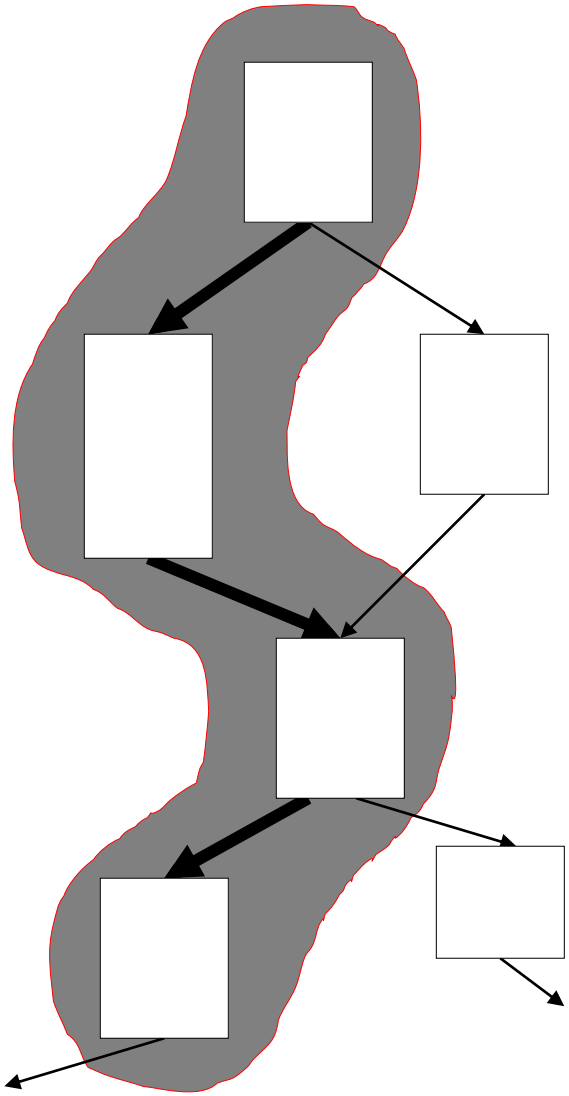
Software pipelining pays startup/wind-down costs only once per loop, not once per iteration

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?

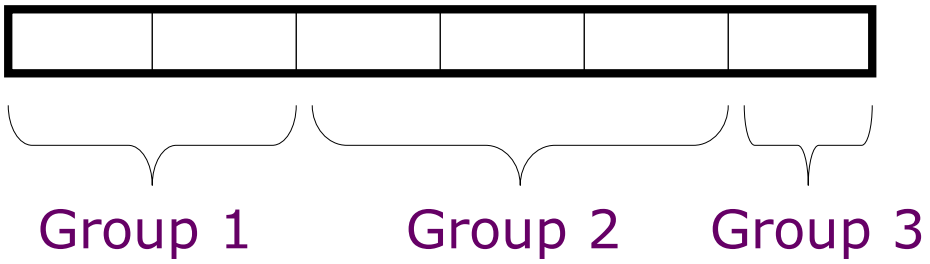
Use profiling feedback or compiler heuristics to find common branch paths

Problems with "Classic" VLIW

- Knowing branch probabilities
  - Profiling requires an 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

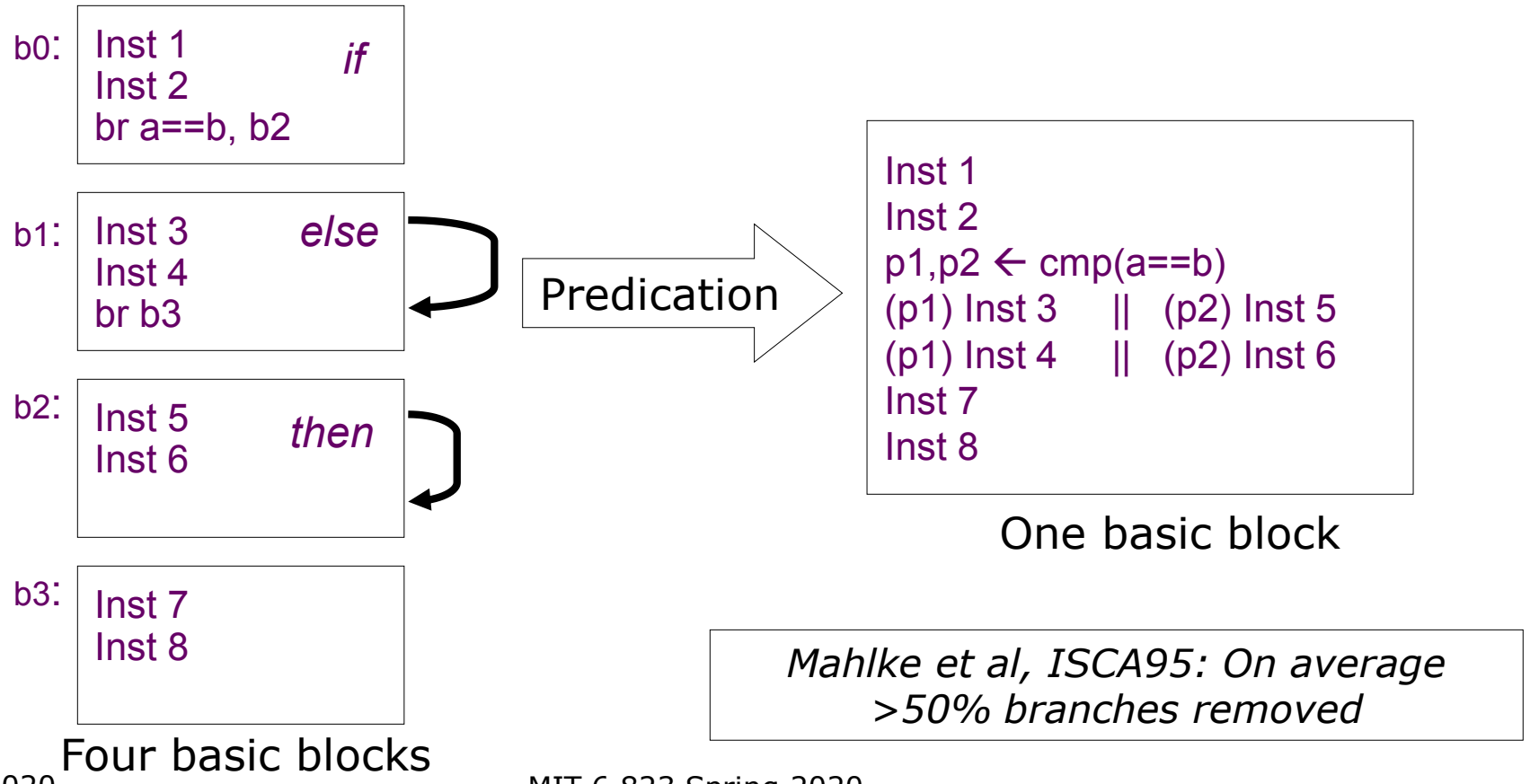
- Hardware ensures that loads take exactly MLR cycles to return values into processor pipeline
  - Hardware buffers loads that return early
  - Hardware stalls processor if loads return late

IA-64 Predicated Execution

Problem: Mispredicted branches limit ILP

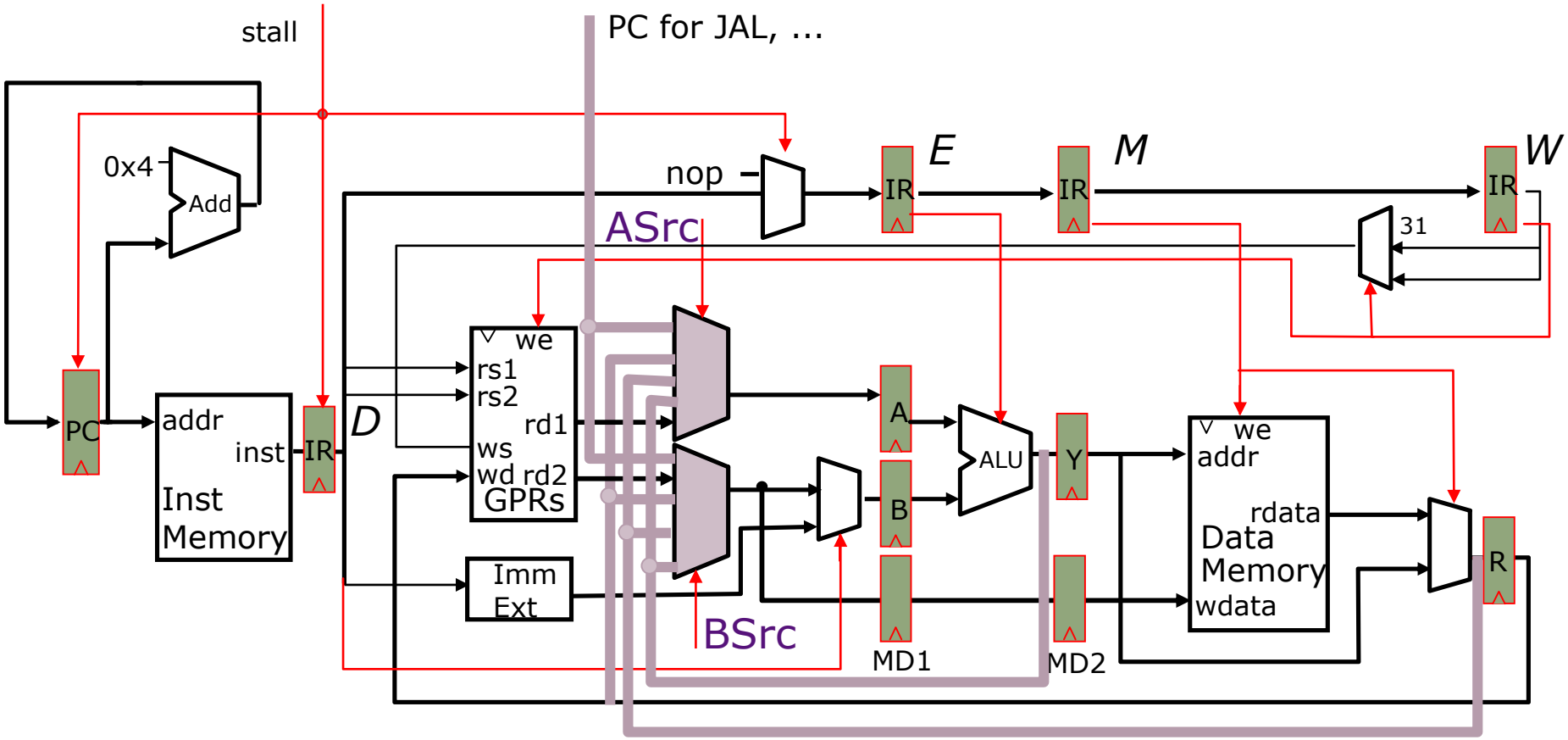
Solution: Eliminate hard-to-predict branches with predicated execution

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



Mahlke et al, ISCA95: On average >50% branches removed

Fully Bypassed Datapath

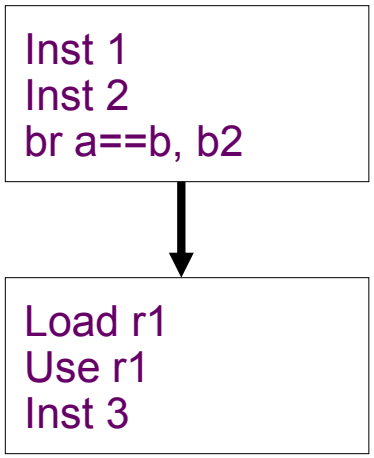


Where does predication fit in?

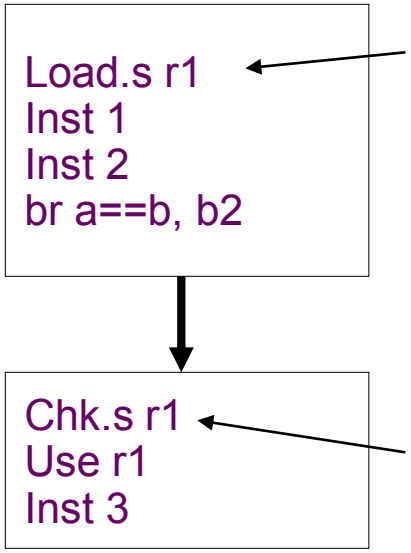
IA-64 Speculative Execution

Problem: Branches restrict compiler code motion

Solution: Speculative operations that don't cause exceptions



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



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

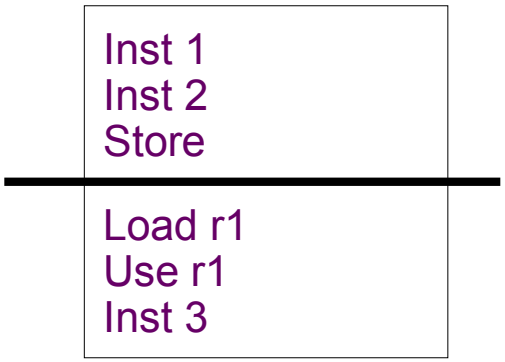
Check for exception in original home block jumps to fixup code if exception detected

Particularly useful for scheduling long latency loads early

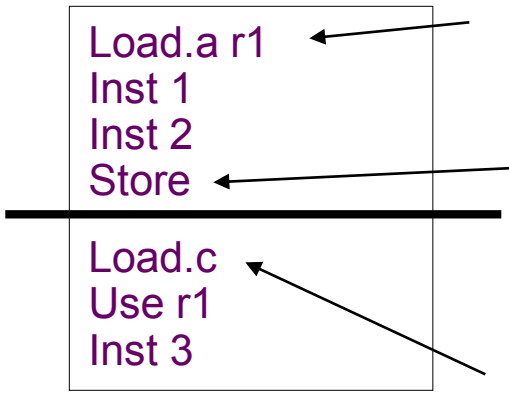
IA-64 Data Speculation

Problem: Possible memory hazards limit code scheduling

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



Can't move load above store because store might be to same address



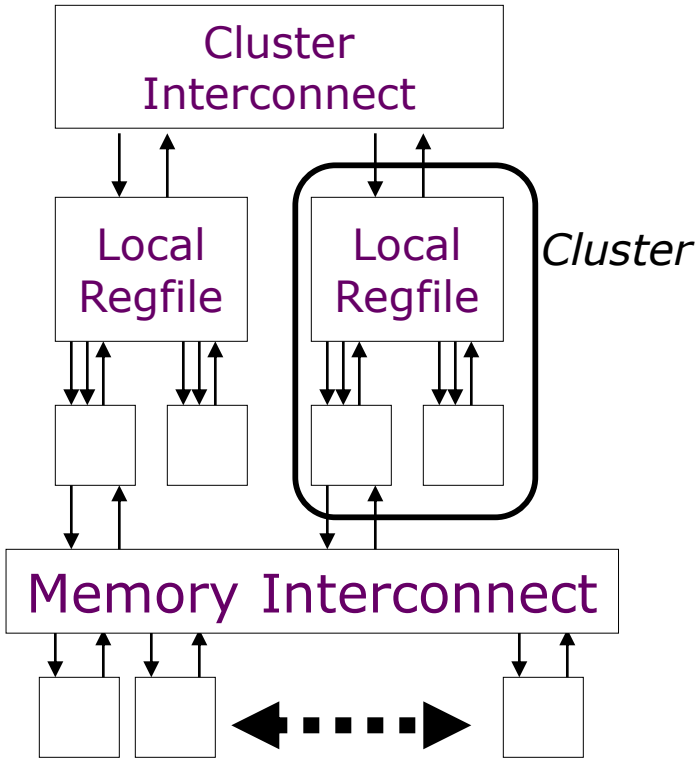
Data speculative load adds address to address check table

Store invalidates any matching loads in address check table

Check if load invalid (or missing), jump to fixup code if so

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!

Next Lecture: Vector Processors