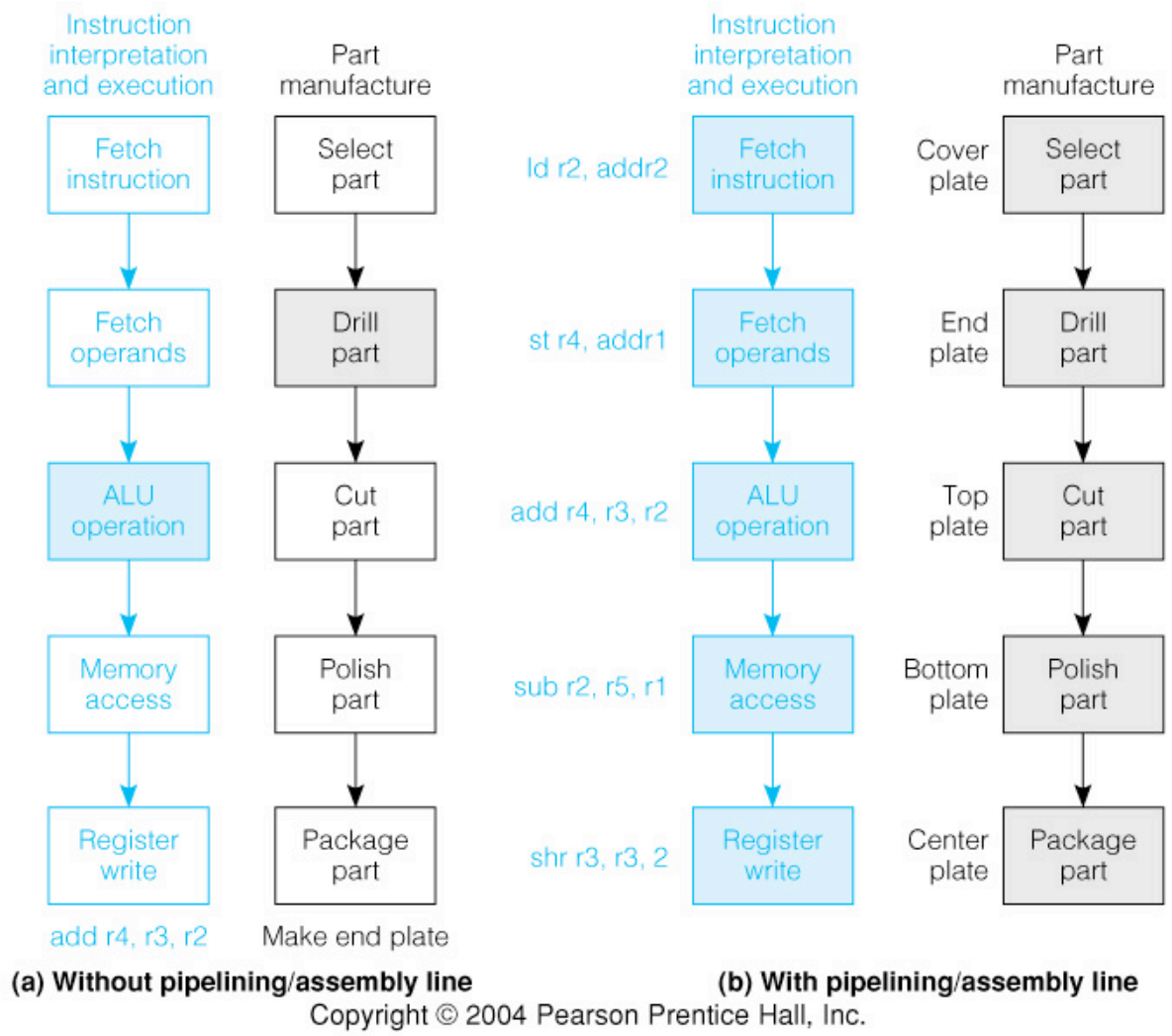# Chapter 5 Overview

- The principles of pipelining

- A pipelined design of SRC

- Pipeline hazards

- Instruction-level parallelism (ILP)

  - Superscalar processors

  - Very Long Instruction Word (VLIW) machines

- Microprogramming

  - Control store and micro-branching

  - Horizontal and vertical microprogramming

Fig 5.1 Executing Machine Instructions vs. Manufacturing Small Parts

(a) Without pipelining/assembly line

Instruction interpretation and execution	Part manufacture
Fetch instruction	Select part
Fetch operands	Drill part
ALU operation	Cut part
Memory access	Polish part
Register write	Package part

add r4, r3, r2   Make end plate

(b) With pipelining/assembly line

	Instruction interpretation and execution		Part manufacture
ld r2, addr2	Fetch instruction	Cover plate	Select part
st r4, addr1	Fetch operands	End plate	Drill part
add r4, r3, r2	ALU operation	Top plate	Cut part
sub r2, r5, r1	Memory access	Bottom plate	Polish part
shr r3, r3, 2	Register write	Center plate	Package part

Copyright © 2004 Pearson Prentice Hall, Inc.

The Pipeline Stages

- 5 pipeline stages are shown
  - 1. Fetch instruction
  - 2. Fetch operands
  - 3. ALU operation
  - 4. Memory access
  - 5. Register write
- 5 instructions are executing
  - `shr r3, r3, 2  ;storing result in r3`
  - `sub r2, r5, r1 ;idle, no mem. access needed`
  - `add r4, r3, r2 ;adding in ALU`
  - `st  r4, addr1  ;accessing r4 and addr1`
  - `ld  r2, addr2  ;instruction being fetched`

Notes on Pipelining Instruction Processing

- Pipeline stages are shown top to bottom in order traversed by one instruction

- Instructions listed in order they are fetched

- Order of insts. in pipeline is reverse of listed

- If each stage takes one clock:

  - every instruction takes 5 clocks to complete

  - some instruction completes every clock tick

- Two performance issues: instruction latency, and instruction bandwidth

Dependence Among Instructions

- Execution of some instructions can depend on the completion of others in the pipeline

- One solution is to "stall" the pipeline
  - early stages stop while later ones complete processing

- Dependences involving registers can be detected and data "forwarded" to instruction needing it, without waiting for register write

- Dependence involving memory is harder and is sometimes addressed by restricting the way the instruction set is used
  - "Branch delay slot" is example of such a restriction
  - "Load delay" is another example

Branch and Load Delay Examples

Branch Delay

```
brz r2, r3
add r6, r7, r8  ←——————  This inst. always executed
st  r6, addr1   ←——————  Only done if r3 ≠ 0
```

Load Delay

```
ld  r2, addr             This inst. gets "old"
add r5, r1, r2  ←——————  value of r2
shr r1,r1,4
sub r6, r8, r2  ←——————  This inst. gets r2 value
                         loaded from addr
```

- Working of instructions not changed, but way they work together is

Characteristics of Pipelined Processor Design

- Main memory must operate in one cycle
  - This can be accomplished by expensive memory, but
  - It is usually done with cache, to be discussed in Chap. 7
- Instruction and data memory must appear separate
  - Harvard architecture has separate instruction & data memories
  - Again, this is usually done with separate caches
- Few buses are used
  - Most connections are point to point
  - Some few-way multiplexers are used
- Data is latched (stored in temporary registers) at each pipeline stage—called "pipeline registers."
- ALU operations take only 1 clock (esp. shift)

Adapting Instructions to Pipelined Execution

- All instructions must fit into a common pipeline stage structure
- We use a 5 stage pipeline for the SRC

  1) Instruction fetch

  2) Decode and operand access

  3) ALU operations

  4) Data memory access

  5) Register write

- We must fit load/store, ALU, and branch instructions into this pattern

Fig 5.2   ALU Instructions fit into 5 Stages

- Second ALU operand comes either from a register or instruction register c2 field

- Op code must be available in stage 3 to tell ALU what to do

- Result register, ra, is written in stage 5

- No memory operation

Instruction memory

PC

1. Instruction fetch

Inc4

IR2 | op, ra | C2(4..0)

Register file
R[rb] R[rc] R[ra]

regwrite

ra

2. Decode and operand read

Mp4

X3   Y3

3. ALU operation

Decode → ALU

Z4

4. Memory access

5. ra write
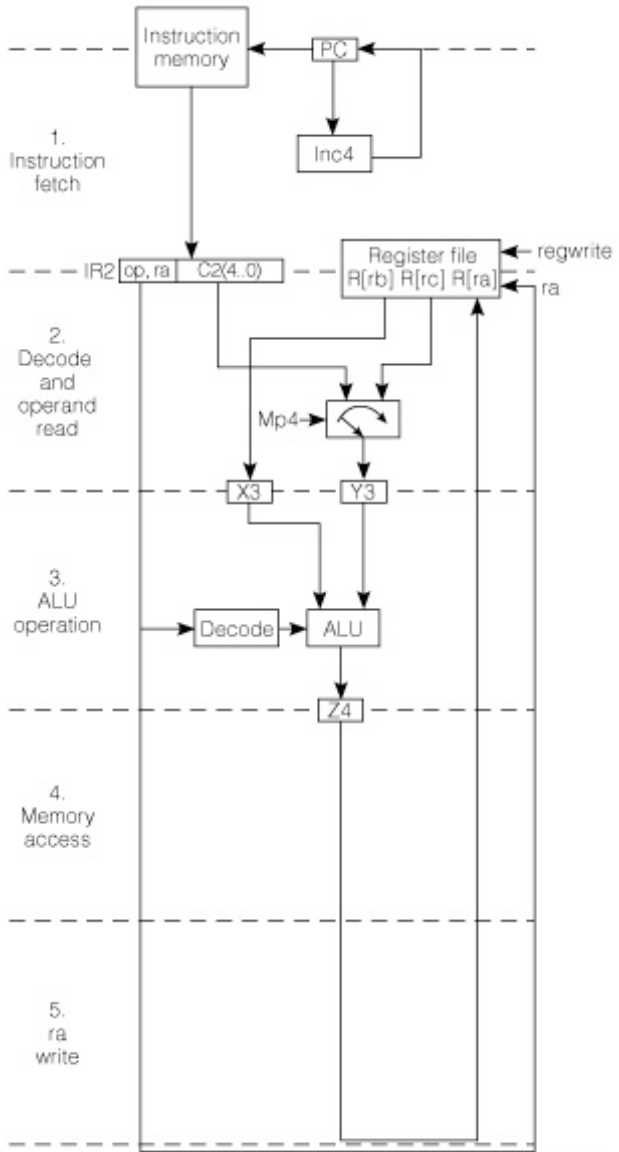
Copyright © 2004 Pearson Prentice Hall, Inc.

Figure 5.3 Logic Expressions Defining Pipeline Stage Activity

branch := br ╱ ∨ brl :

cond := (IR2⟨2..0⟩ = 1) ∨ ((IR2⟨2..1⟩=1)∧(IR2⟨0⟩⊕R[rb]=0)) ∨

                                    ((IR2⟨2..1⟩=2)∧(IR2⟨0⟩⊕R[rb]⟨31⟩) :

sh := shr ∨ shra ∨ shl ∨ shc :

alu := add ∨ addi ∨ sub ∨ neg ∨ and ∨ andi∨ or ∨ ori ∨ not ∨ sh :

imm := addi ∨ andi ∨ ori ∨ (sh ∧ (IR2⟨4..0⟩ ≠ 0) :

load := ld ∨ ldr :

ladr := la ∨ lar :

store := st ∨ str :

l-s := load ∨ ladr ∨ store :

regwrite := load ∨ ladr ∨ brl ∨ alu: ;these instructions write the register file

dsp := ld ∨ st ∨ la :   ;instructions that use disp addressing

rl := ldr ∨ str ∨ lar :   ;instructions that use rel addressing

Notes on the Equations and Different Stages

- The logic equations are based on the instruction in the stage where they are used

- When necessary, we append a digit to a logic signal name to specify it is computed from values in that stage

- Thus regwrite5 is true when the opcode in stage 5 is load5 ∨ ladr5 ∨ brl5 ∨ alu5, all of which are determined from op5

Fig 5.4 Load and Store Instructions

- ALU computes effective addresses
- Stage 4 does read or write
- Result reg. written only on load
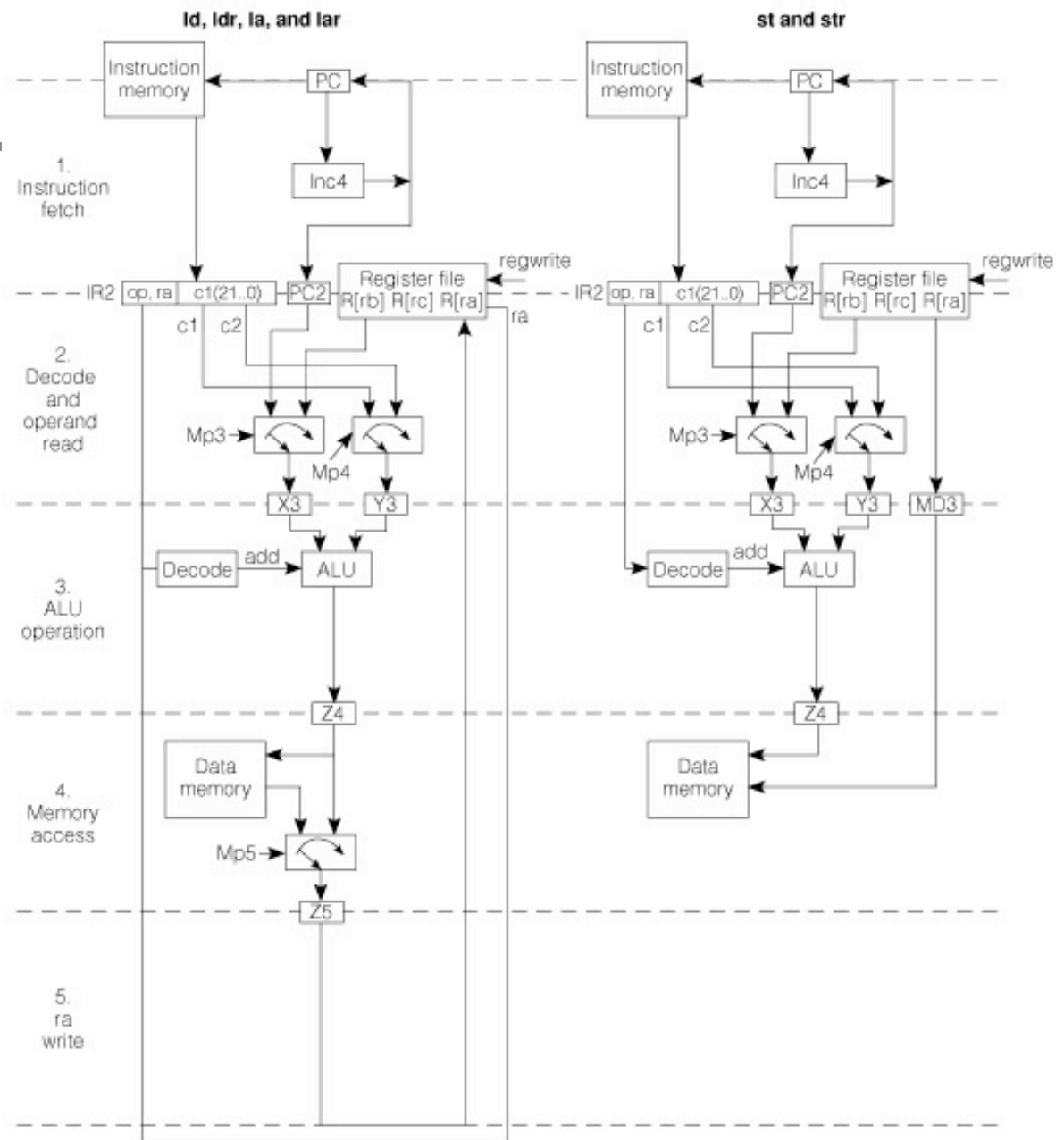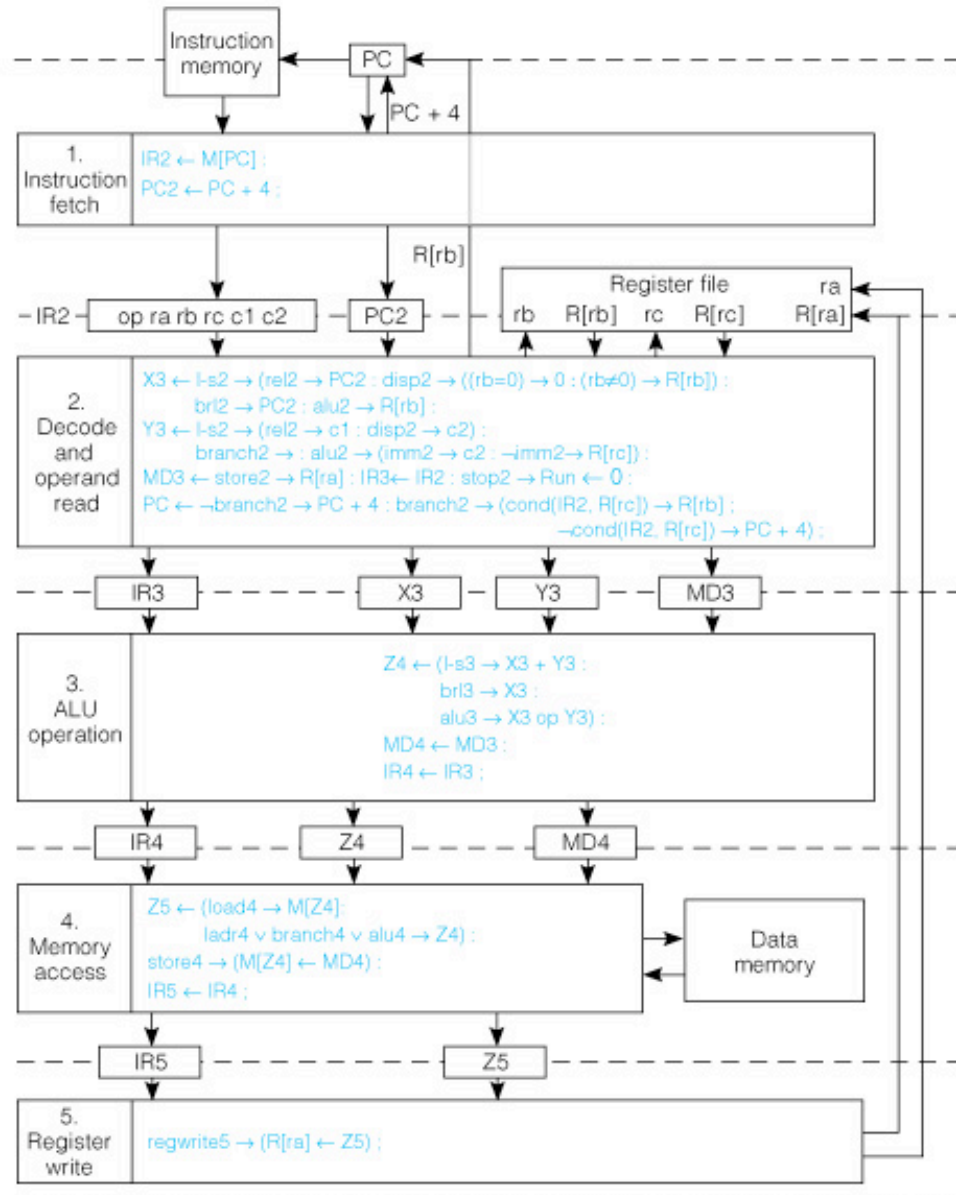
Fig 5.5 The Branch Instructions

- The new program counter value is known in stage 2

  —but not in stage 1

- Only branch&link does a register write in stage 5

- There is no ALU or memory operation

Branch br and brl

1. Instruction fetch

2. Decode and operand read

3. ALU operation

4. Memory access

5. ra write
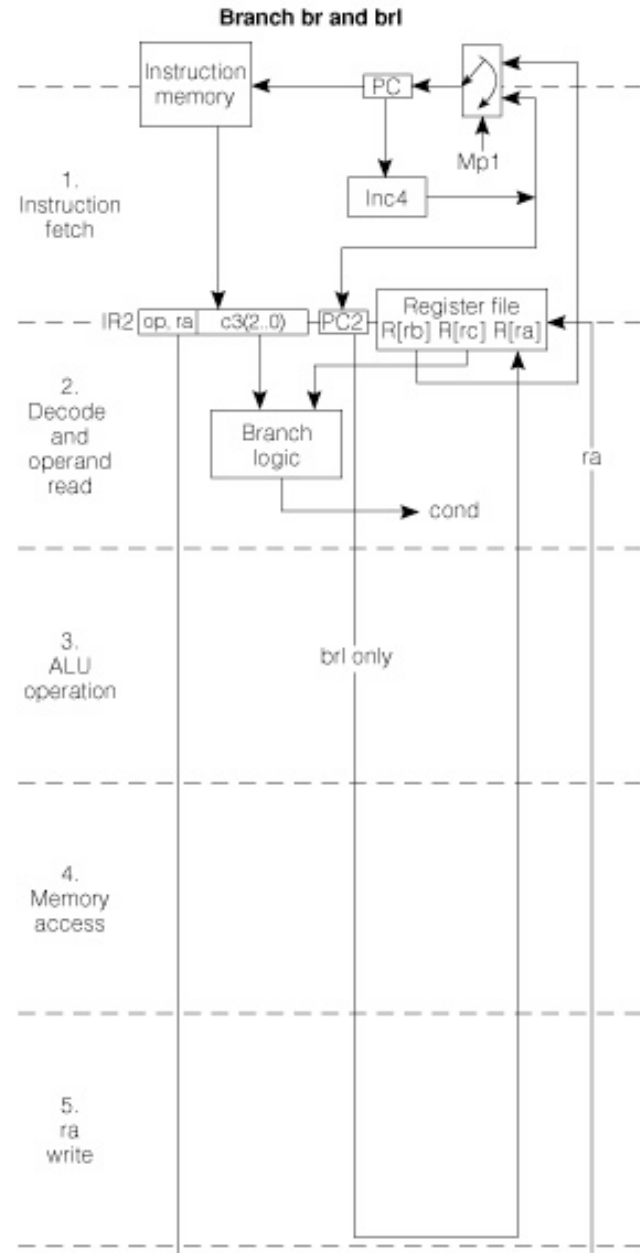
Fig 5.6 SRC Pipeline Registers and RTN Specification

- The pipeline registers pass info. from stage to stage
- RTN specifies output reg. values in terms of input reg. values for stage
- Discuss RTN at each stage on blackboard

Instruction memory   PC
PC + 4

1. Instruction fetch
IR2 ← M[PC] ;
PC2 ← PC + 4 ;

R[rb]

Register file   ra

IR2 — op ra rb rc c1 c2 — PC2 —— rb R[rb] rc R[rc] R[ra]

2. Decode and operand read
X3 ← I-s2 → (rel2 → PC2 : disp2 → ((rb=0) → 0 : (rb≠0) → R[rb]) :
    brl2 → PC2 : alu2 → R[rb] :
Y3 ← I-s2 → (rel2 → c1 : disp2 → c2) :
    branch2 → : alu2 → (imm2 → c2 : ¬imm2→ R[rc]) :
MD3 ← store2 → R[ra] : IR3← IR2 : stop2 → Run ← 0 :
PC ← ¬branch2 → PC + 4 : branch2 → (cond(IR2, R[rc]) → R[rb] :
    ¬cond(IR2, R[rc]) → PC + 4) ;

IR3   X3   Y3 — — MD3

3. ALU operation
Z4 ← (I-s3 → X3 + Y3 :
    brl3 → X3 :
    alu3 → X3 op Y3) :
MD4 ← MD3 :
IR4 ← IR3 ;

IR4   Z4   MD4

4. Memory access
Z5 ← (load4 → M[Z4] :
    ladr4 v branch4 v alu4 → Z4) :
store4 → (M[Z4] ← MD4) :
IR5 ← IR4 ;

Data memory

IR5   Z5

5. Register write
regwrite5 → (R[ra] ← Z5) ;

Copyright © 2004 Pearson Prentice Hall, Inc.

Global State of the Pipelined SRC

- PC, the general registers, instruction memory, and data memory is the global machine state

- PC is accessed in stage 1 (& stage 2 on branch)

- Instruction memory is accessed in stage 1

- General registers are read in stage 2 and written in stage 5
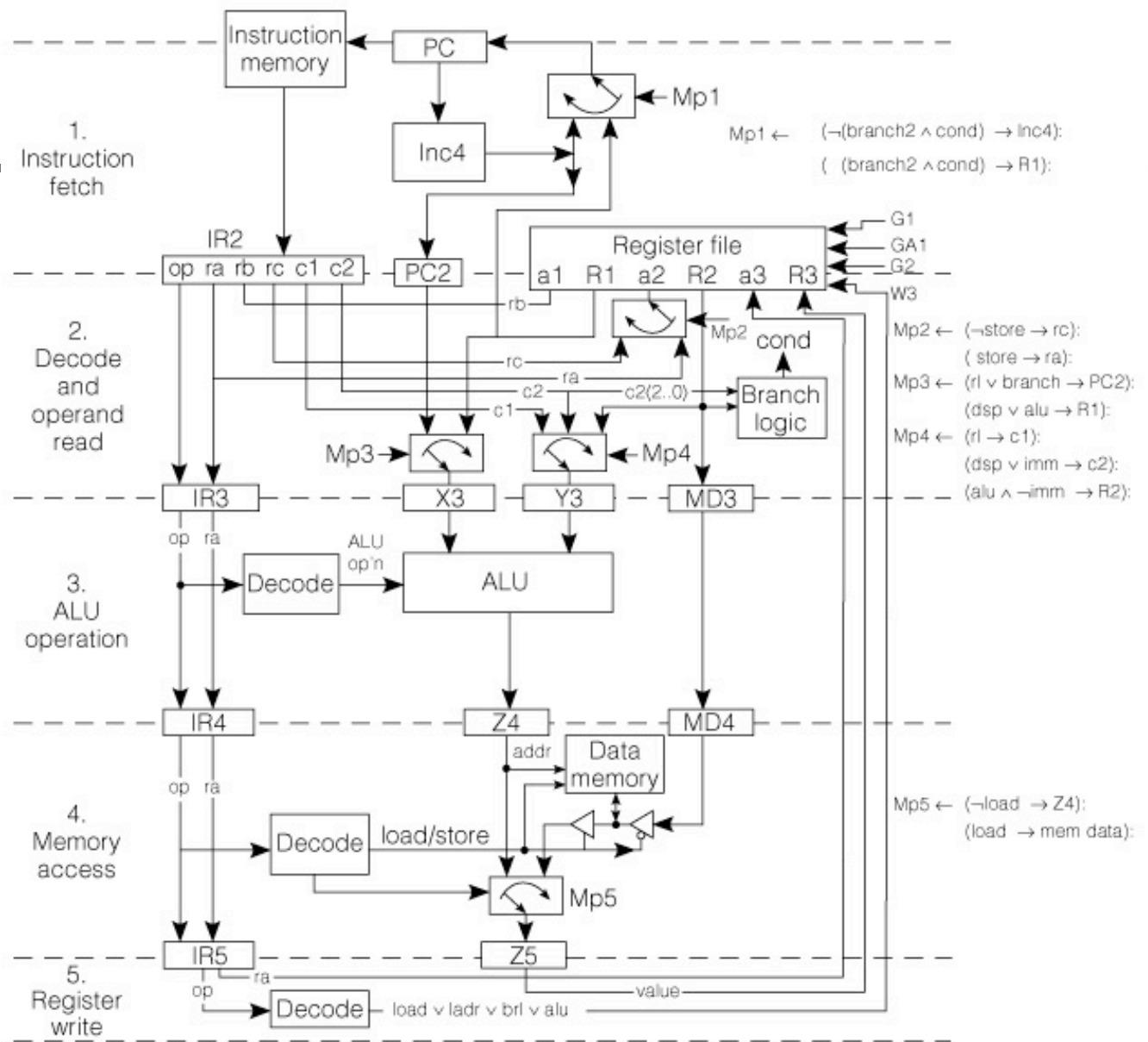
- Data memory is only accessed in stage 4

Restrictions on Access to Global State by Pipeline

- We see why separate instruction and data memories (or caches) are needed

- When a load or store accesses data memory in stage 4, stage 1 is accessing an instruction

    - Thus two memory accesses occur simultaneously

- Two operands may be needed from registers in stage 2 while another instruction is writing a result register in stage 5

    - Thus as far as the registers are concerned, 2 reads and a write happen simultaneously

- Increment of PC in stage 1 must be overridden by a successful branch in stage 2

Fig 5.7 Pipeline Data Path & Control Signals

- Most control signals shown and given values
- Multiplexer control is stressed in this figure

1. Instruction fetch

Instruction memory — PC

Mp1 ← (¬(branch2 ∧ cond) → Inc4):
      ((branch2 ∧ cond) → R1):

2. Decode and operand read

IR2 op ra rb rc c1 c2   PC2   a1 R1 a2 R2 a3 R3   Register file   G1 GA1 G2 W3

Mp2 cond   Branch logic

Mp2 ← (¬store → rc):
      (store → ra):
Mp3 ← (rl ∨ branch → PC2):
      (dsp ∨ alu → R1):
Mp4 ← (rl → c1):
      (dsp ∨ imm → c2):
      (alu ∧ ¬imm → R2):

Mp3 →   Mp4

3. ALU operation

IR3   X3   Y3   MD3

op ra   ALU op'n   Decode   ALU

4. Memory access

IR4   Z4   MD4

op ra   Decode load/store   addr Data memory   Mp5

Mp5 ← (¬load → Z4):
      (load → mem data):

5. Register write

IR5   Z5

op ra   Decode   load ∨ ladr ∨ brl ∨ alu   value

Example of Propagation of Instructions Through Pipe

```
100:   add     r4, r6, r8;           R[4] ← R[6] + R[8];
104:   ld      r7, 128(r5);          R[7] ← M[R[5]+128];
108:   brl     r9, r11, 001;         PC ← R[11]: R[9] ← PC;
112:   str     r12, 32;              M[PC+32] ← R[12];

       . . . . . .

512:   sub     ...                   next instruction
```

- It is assumed that R[11] contains 512 when the brl instruction is executed

- R[6] = 4 and R[8] = 5 are the add operands

- R[5] =16 for the ld and R[12] = 23 for the str

Fig 5.8 Cycle 1 add Enters Pipe

- Program counter is incremented to 104

512: sub   ...

. . . . . .

112: str   r12, #32
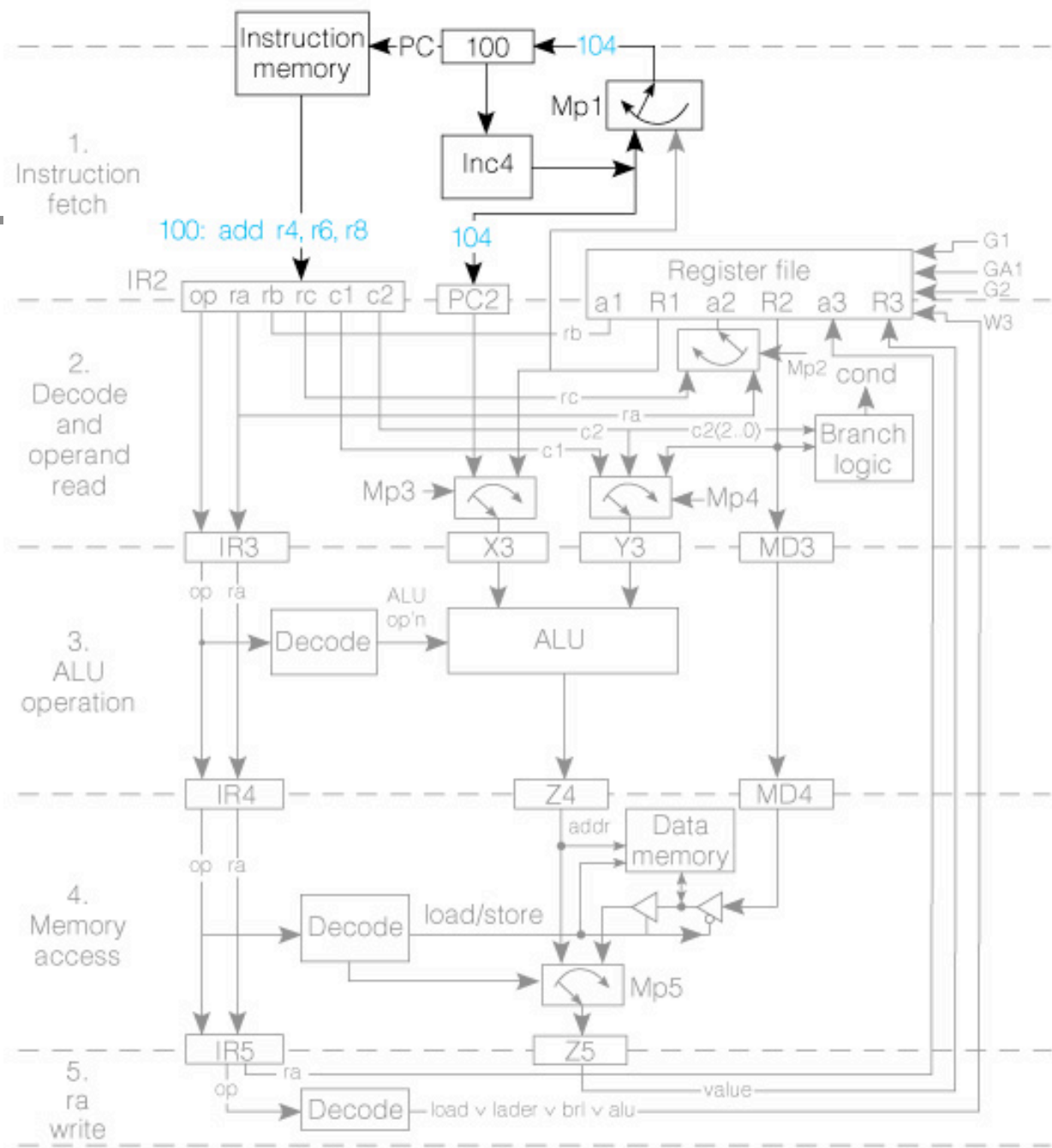
108: brl   r9, r11, 001

104: ld   r7, r5, #128

100: add   r4, r6, r8

- add operands are fetched in stage 2

512: sub ...
**   . . . . . .**
112: str   r12, #32
108: brl   r9, r11, 001
104: ld   r7, r5, #128
100: add   r4, r6, r8

Copyright © 2004 Pearson Prentice Hall, Inc.
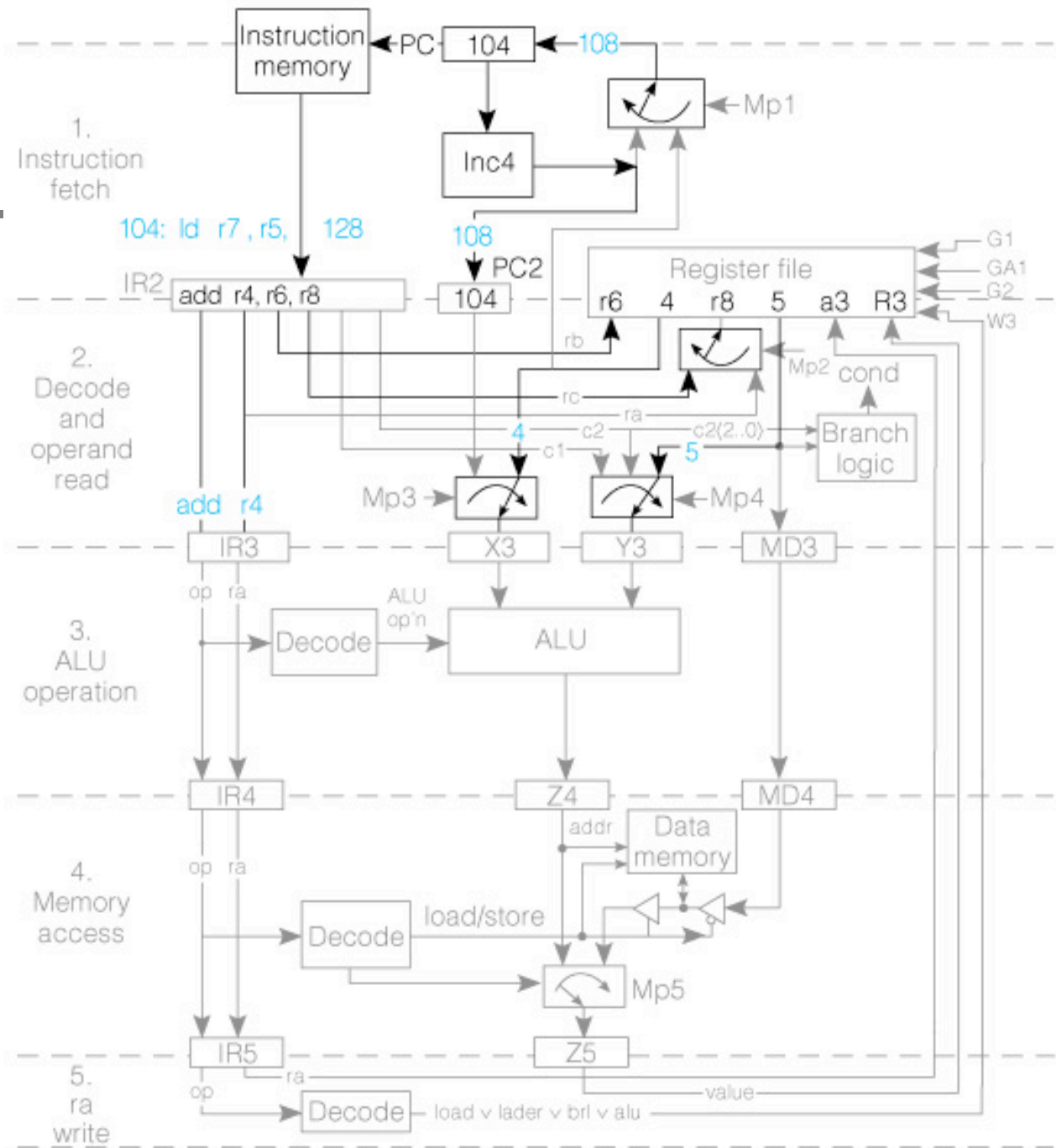
Computer Systems Design and Architecture

Fig 5.10 Cycle 3 brl Enters Pipe

- add performs its arithmetic in stage 3

512: sub ...
. . . . . .
112: str   r12, #32
108: brl   r9, r11, 001
104: ld   r7, r5, #128
100: add   r4, r6, r8
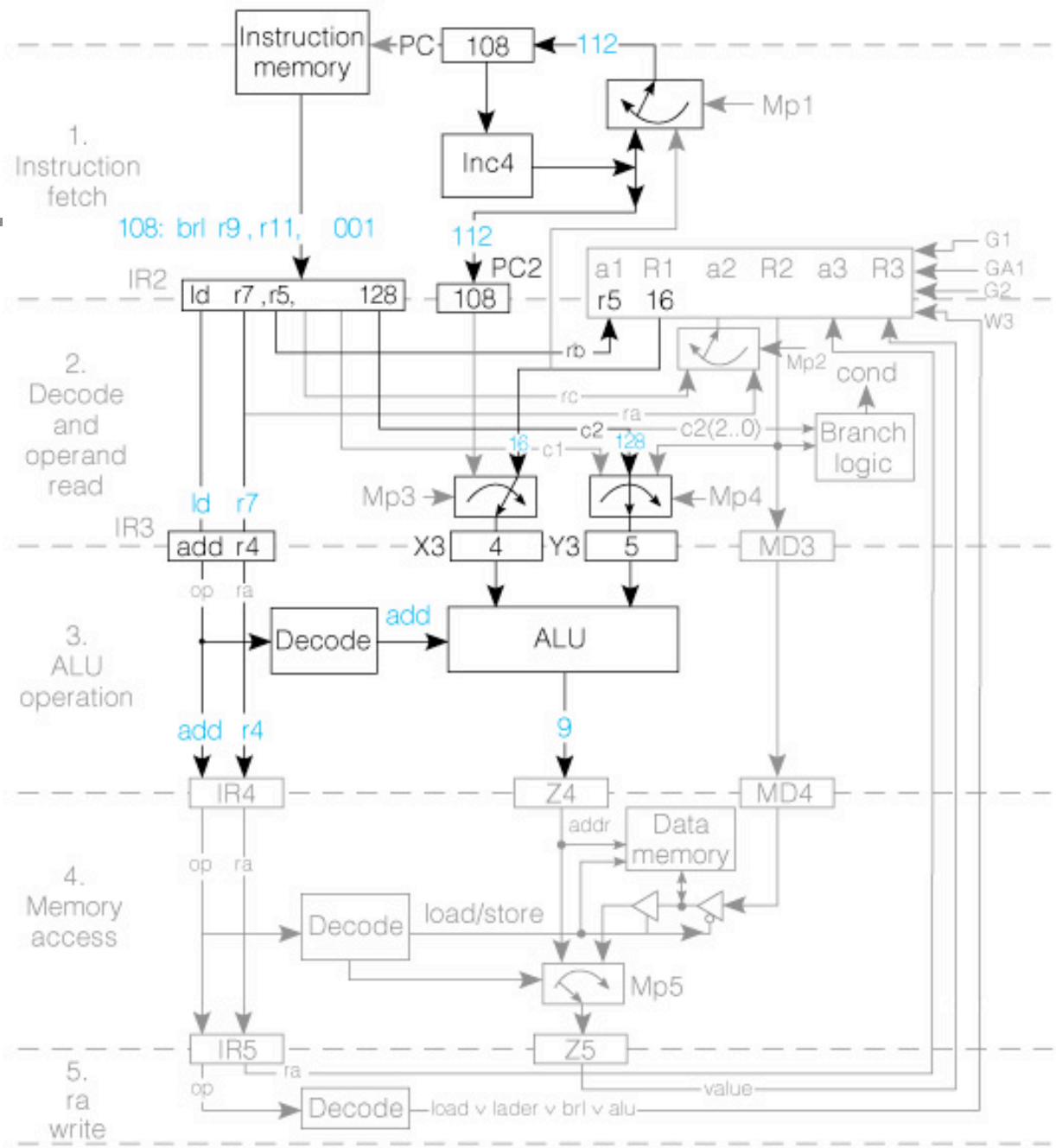
Copyright © 2004 Pearson Prentice Hall, Inc.

Fig 5.11 Cycle 4
str enters pipe

- add is idle in stage 4
- Success of brl changes program counter to 512

512: sub ...

. . . . . .
112: str   r12, #32
108: brl   r9, r11, 001
104: ld   r7, r5, #128
100: add   r4, r6, r8

Copyright © 2004 Pearson Prentice Hall, Inc.
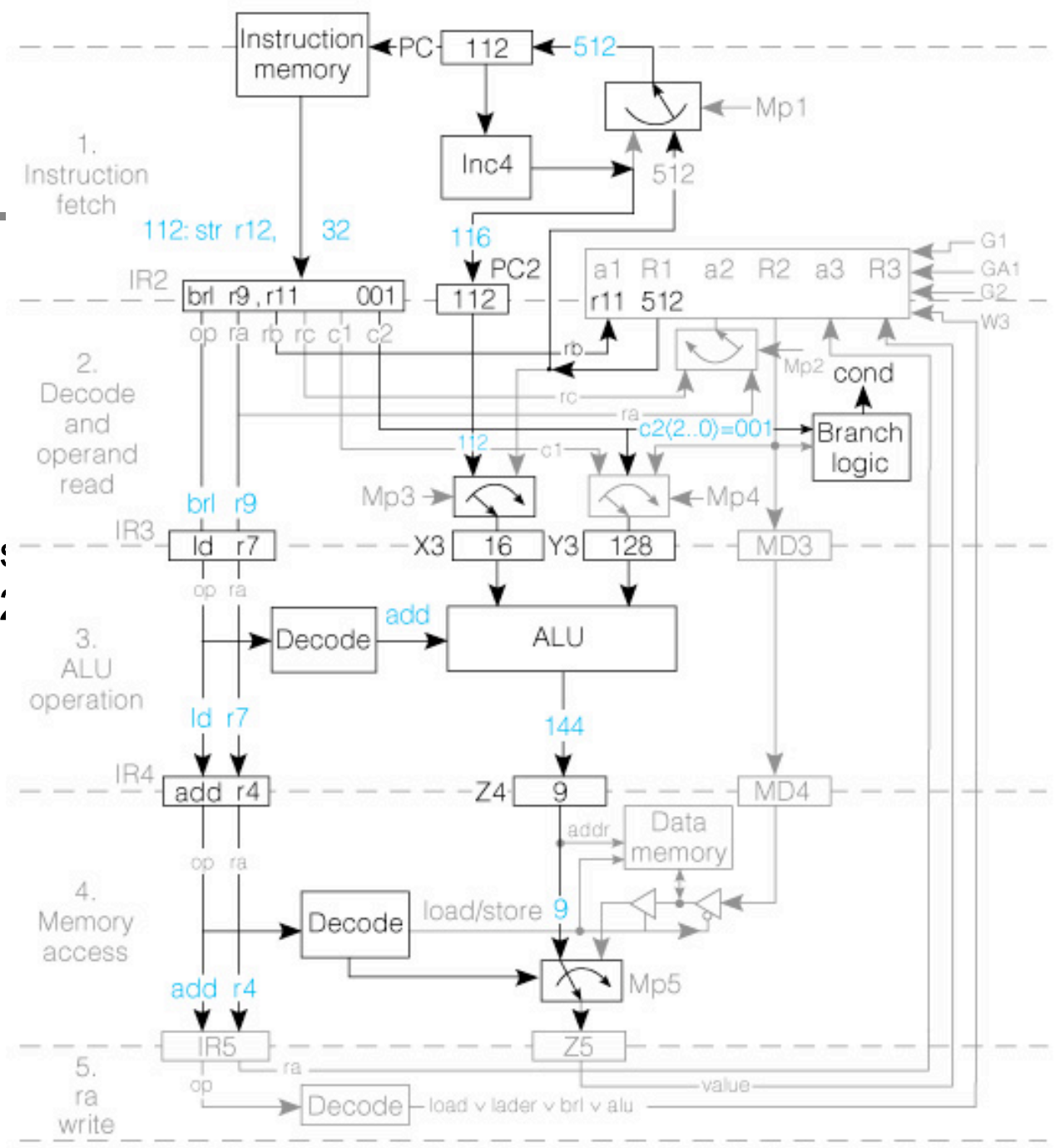
Fig 5.12 Cycle 5
sub Enters Pipe

- add completes in stage 5
- sub is fetched from loc. 512 after successful brl

512: sub ...

      . . . . . .
112: str   r12, #32
108: brl   r9, r11, 001
104: ld   r7, r5, #128
100: add r4, r6, r8
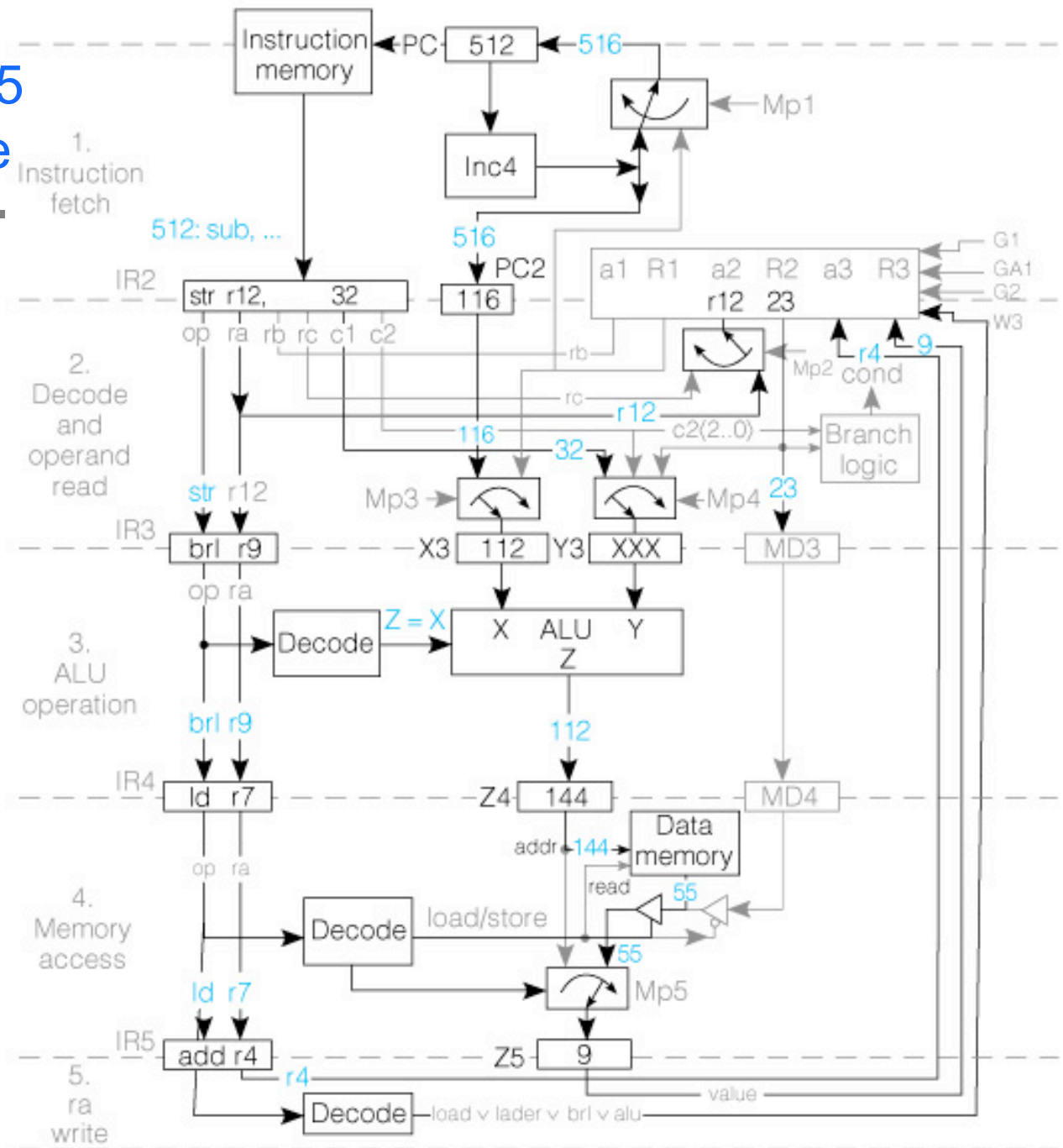
Functions of the Pipeline Registers in SRC

- Registers between stages 1 & 2:
  - I2 holds full instruction including any reg. fields and constant;
  - PC2 holds the incremented PC from instruction fetch
- Registers between stages 2 & 3:
  - I3 holds op code and ra (needed in stage 5);
  - X3 holds PC or a reg. value (for link or 1st ALU operand);
  - Y3 holds c1 or c2 or a reg. value as 2nd ALU operand;
  - MD3 is used for a register value to be stored in mem.
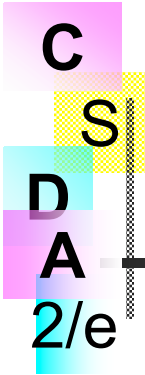
Functions of the Pipeline Registers in SRC (continued)

- Registers between stages 3 & 4:
    - I4 has op code and ra;
    - Z4 has mem. address or result reg. value
    - MD4 has value to be stored in data memory
- Registers between stages 4 & 5:
    - I5 has op code and destination register number, ra
    - Z5 has value to be stored in destination register: from ALU result, PC link value, or fetched data

Functions of the SRC Pipeline Stages

- Stage 1: fetches instruction
  - PC incremented or replaced by successful branch in stage 2
- Stage 2: decodes inst. and gets operands
  - Load or store gets operands for address computation
  - Store gets register value to be stored as 3rd operand
  - ALU operation gets 2 registers or register and constant
- Stage 3: performs ALU operation
  - Calculates effective address or does arithmetic/logic
  - May pass through link PC or value to be stored in mem.

Functions of the SRC Pipeline Stages (continued)

- Stage 4: accesses data memory
  - Passes Z4 to Z5 unchanged for non-memory instructions
  - Load fills Z5 from memory
  - Store uses address from Z4 and data from MD4(no longer needed)
- Stage 5: writes result register
  - Z5 contains value to be written, which can be ALU result, effective address, PC link value, or fetched data
  - ra field always specifies result register in SRC

Dependence Between Instructions in Pipe: Hazards

- Instructions that occupy the pipeline together are being executed in parallel

- This leads to the problem of instruction dependence, well known in parallel processing

- The basic problem is that an instruction depends on the result of a previously issued instruction that is not yet complete

- Two categories of hazards

Data hazards: incorrect use of old and new data

Branch hazards: fetch of wrong instruction on a change in PC

General Classification of Data Hazards
(Not Specific to SRC)

- A read after write hazard (RAW) arises from a <u>flow dependence</u>, where an instruction uses data produced by a previous one

- A write after read hazard (WAR) comes from an <u>anti-dependence</u>, where an instruction writes a new value over one that is still needed by a previous instruction

- A write after write hazard (WAW) comes from an <u>output dependence</u>, where two parallel instructions write the same register and must do it in the order in which they were issued

Detecting Hazards and Dependence Distance

- To detect hazards, pairs of instructions must be considered

- Data is normally available after being written to reg.

- Can be made available for forwarding as early as the stage where it is produced

  - Stage 3 output for ALU results, stage 4 for mem. fetch

- Operands normally needed in stage 2

- Can be received from forwarding as late as the stage in which they are used

  - Stage 3 for ALU operands and address modifiers, stage 4 for stored register, stage 2 for branch target

Data Hazards in SRC

- Since all data memory access occurs in stage 4, memory writes and reads are sequential and give rise to no hazards

- Since all registers are written in the last stage, WAW and WAR hazards do not occur

  - Two writes always occur in the order issued, and a write always follows a previously issued read

- SRC hazards on register data are limited to RAW hazards coming from flow dependence

- Values are written into registers at the end of stage 5 but may be needed by a following instruction at the beginning of stage 2

Possible Solutions to the Register Data Hazard Problem

- Detection:
    - The machine manual could list rules specifying that a dependent instruction cannot be issued less than a given number of steps after the one on which it depends
    - This is usually too restrictive
    - Since the operation and operands are known at each stage, dependence on a following stage can be detected
- Correction:
    - The dependent instruction can be "stalled" and those ahead of it in the pipeline allowed to complete
    - Result can be "forwarded" to a following inst. in a previous stage without waiting to be written into its register
- Preferred SRC design will use detection, forwarding and stalling only when unavoidable

RAW, WAW, and WAR Hazards

RAW	WAW	WAR
1. add r0, r1, r2	1. add r0, r1, r2	1. add r2, r1, r0
2. sub r4, r3, r0	2. sub r0, r4, r5	2. sub r0, r3, r4

- RAW hazards are due to causality: one cannot use a value before it has been produced.

- WAW and WAR hazards can only occur when instructions are executed in parallel or out of order.
  - Not possible in SRC.
  - Are only due to the fact that registers have the same name.
  - Can be fixed by renaming one of the registers or by delaying the updating of a register until the appropriate value has been produced.

Tbl 5.1 Instruction Pair Hazard Interaction

Write to Reg. File

Result Normally/Earliest available

Read from Reg. File

Value
Normally/
Latest
needed

Class	N/L	Class	alu	load	ladr	brl
		N/E	6/4	6/5	6/4	6/2
alu	2/3		4/1	4/2	4/1	4/1
load	2/3		4/1	4/2	4/1	4/1
ladr	2/3		4/1	4/2	4/1	4/1
store	2/3		4/1	4/2	4/1	4/1
branch	2/2		4/2	4/3	4/2	4/1

Instruction separation to eliminate hazard, Normal/Forwarded

- Latest needed stage 3 for store is based on address modifier register. The stored value is not needed until stage 4
- Store also needs an operand from ra. See Text Tbl 5.
- Instruction separation is used rather than bubbles because of the applicability to multi-issue, multi-pipelined machines.

Delays Unavoidable by Forwarding

- In the column headed by load, we see the value loaded cannot be available to the next instruction, even with forwarding
  - Can restrict compiler not to put a dependent instruction in the next position after a load (next 2 positions if the dependent instruction is a branch)
- Target register cannot be forwarded to branch from the immediately preceding instruction
  - Code is restricted so that branch target must not be changed by instruction preceding branch (previous 2 instructions if loaded from mem.)
  - Do not confuse this with the branch delay slot, which is a dependence of instruction fetch on branch, not a dependence of branch on something else

Stalling the Pipeline on Hazard Detection

- Assuming hazard detection, the pipeline can be stalled by inhibiting earlier stage operation and allowing later stages to proceed

- A simple way to inhibit a stage is a pause signal that turns off the clock to that stage so none of its output registers are changed

- If stages 1 & 2, say, are paused, then something must be delivered to stage 3 so the rest of the pipeline can be cleared

- Insertion of nop into the pipeline is an obvious choice

Example of Detecting ALU Hazards and Stalling Pipeline

- The following expression detects hazards between ALU instructions in stages 2 & 3 and stalls the pipeline

$(alu3 \wedge alu2 \wedge ((ra3=rb2) \vee (ra3=rc2) \wedge \neg imm2)) \rightarrow$
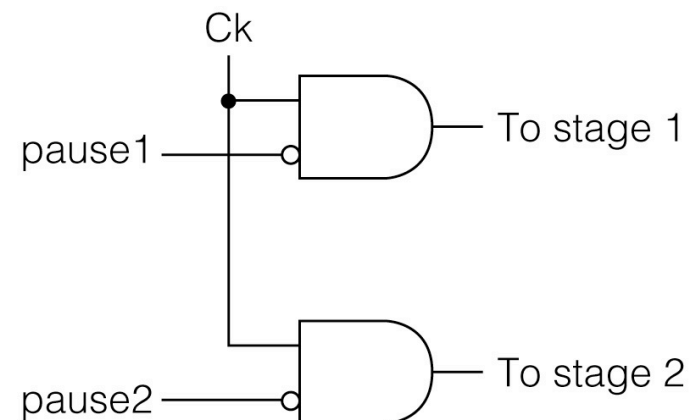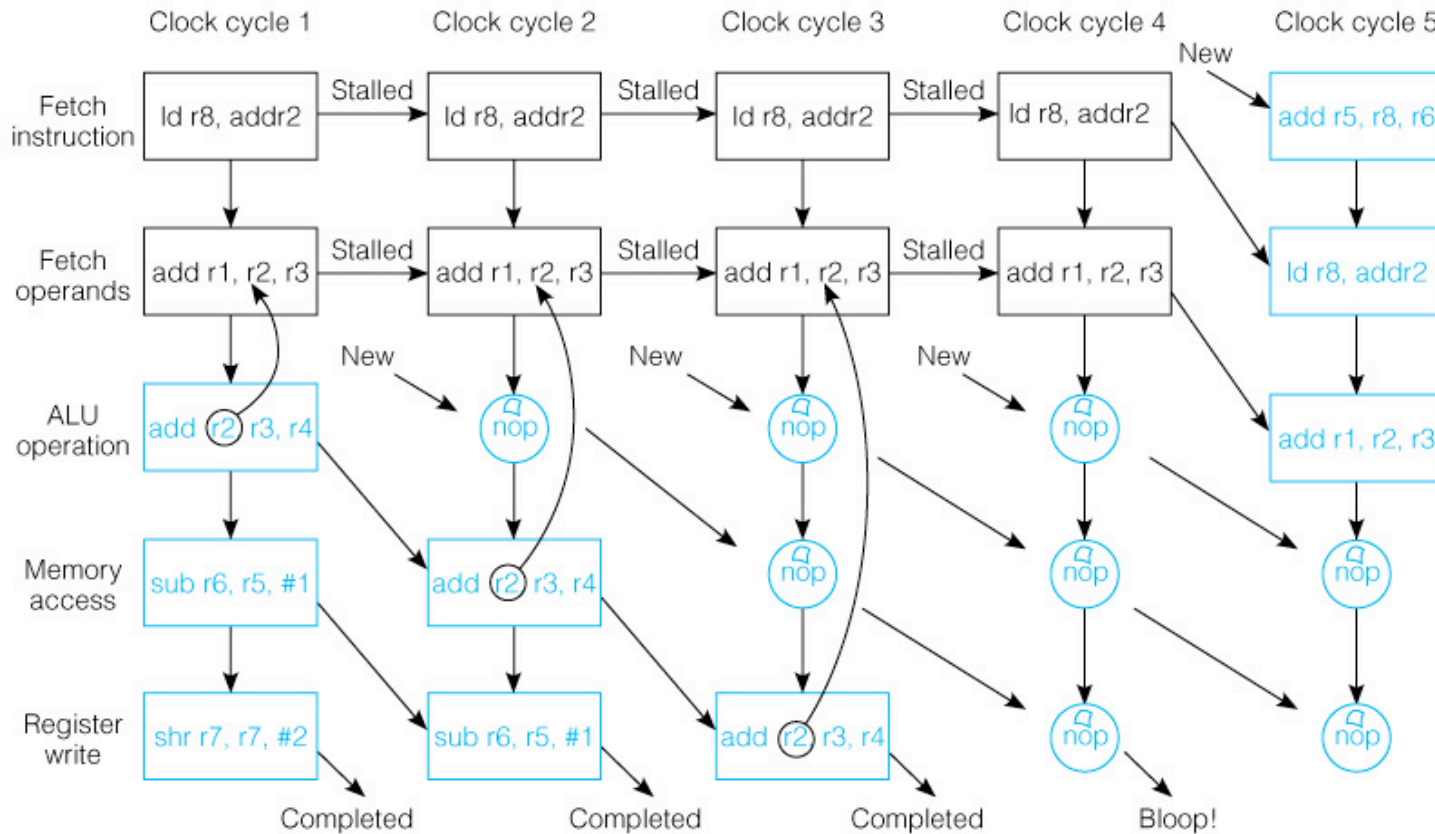 $(pause2: pause1: op3 \leftarrow 0):$

- After such a stall, the hazard will be between stages 2 & 4, detected by

$(alu4 \wedge alu2 \wedge ((ra4=rb2) \vee (ra4=rc2) \wedge \neg imm2)) \rightarrow$
    $(pause2: pause1: op3 \leftarrow 0):$

- Hazards between stages 2 & 5 require

$(alu5 \wedge alu2 \wedge ((ra5=rb2) \vee (ra5=rc2) \wedge \neg imm2)) \rightarrow$
    $(pause2: pause1: op3 \leftarrow 0):$

Fig 5.13

Ck

pause1 ——— To stage 1

pause2 ——— To stage 2

Fig 5.14 Stall Due to a Dependence Between Two alu Instructions

Copyright © 2004 Pearson Prentice Hall, Inc.

Data Forwarding:
from alu Instruction to alu Instruction

- The pair table for data dependencies says that if forwarding is done, dependent alu instructions can be adjacent, not 4 apart

- For this to work, dependences must be detected and data sent from where it is available directly to X or Y input of ALU

- For a dependence of an alu inst. in stage 3 on an alu inst. in stage 5 the equation is

$$alu5 \wedge alu3 \rightarrow ((ra5=rb3) \rightarrow X3 \leftarrow Z5:$$
$$(ra5=rc3) \wedge \neg imm3 \rightarrow Y3 \leftarrow Z5):$$

Data Forwarding:
alu to alu Instruction (continued)

- For an alu inst. in stage 3 depending on one in stage 4, the equation is

$$\text{alu4} \wedge \text{alu3} \rightarrow ((\text{ra4=rb3}) \rightarrow \text{X3} \leftarrow \text{Z4}:$$
$$(\text{ra4=rc3}) \wedge \neg \text{imm3} \rightarrow \text{Y3} \leftarrow \text{Z4}):$$

- We can see that the rb and rc fields must be available in stage 3 for hazard detection

- Multiplexers must be put on the X and Y inputs to the ALU so that Z4 or Z5 can replace either X3 or Y3 as inputs

Fig 5.15 alu to alu Data Forwarding Hardware

- Can be from either Z4 or Z5 to either X or Y input to ALU
- rb & rc needed in stage 3 for detection
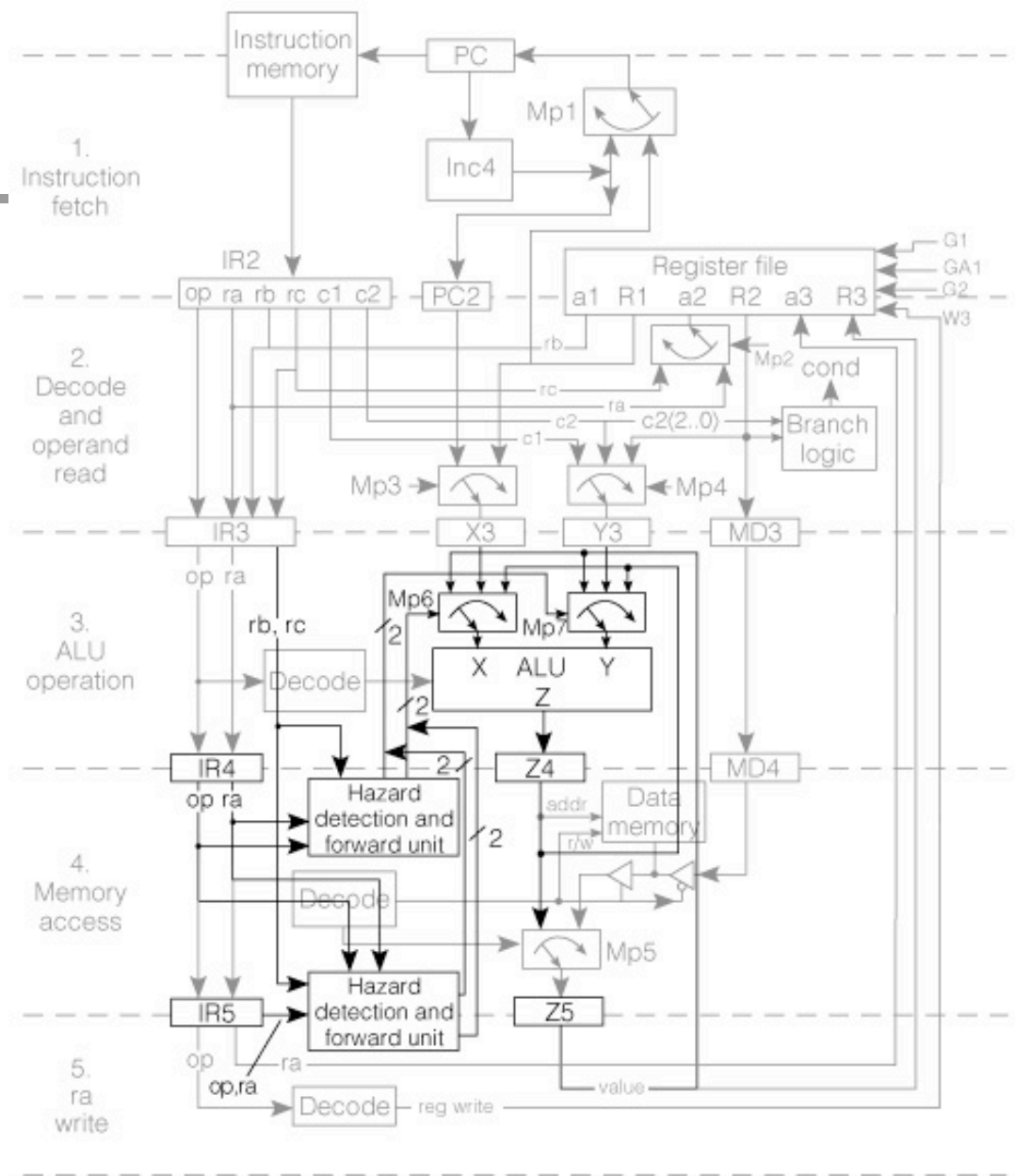
Restrictions Left If Forwarding Done Wherever Possible

1) Branch delay slot

- The instruction after a branch is always executed, whether the branch succeeds or not.

2) Load delay slot

- A register loaded from memory cannot be used as an operand in the next instruction.

- A register loaded from memory cannot be used as a branch target for the next two instructions.

3) Branch target

- Result register of alu or ladr instruction cannot be used as branch target by the next instruction.

br r4
add . . .
. . .

ld r4, 4(r5)
nop
neg r6, r4

ld r0, 1000
nop
nop
br r0

not r0, r1
nop
br r0

Questions for Discussion

- How and when would you debug this design?

- How does RTN and similar Hardware Description Languages fit into testing and debugging?

- What tools would you use, and which stage?

- What kind of software test routines would you use?

- How would you correct errors at each stage in the design?

Instruction Level Parallelism

- A pipeline that is full of useful instructions completes at most one every clock cycle

    Sometimes called the Flynn limit

- If there are multiple function units and multiple instructions have been fetched, then it is possible to start several at once

- Two approaches are: superscalar
    - Dynamically issue as many prefetched instructions to idle function units as possible

- and Very Long Instruction Word (VLIW)
    - Statically compile long instruction words with many operations in a word, each for a different function unit
    - Word size may be 128 or 256 or more bits.

Character of the Function Units in Multiple Issue Machines

- There may be different types of function units
  - Floating point
  - Integer
  - Branch
- There can be more than one of the same type
- Each function unit is itself pipelined
- Branches become more of a problem
  - There are fewer clock cycles between branches
  - Branch units try to predict branch direction
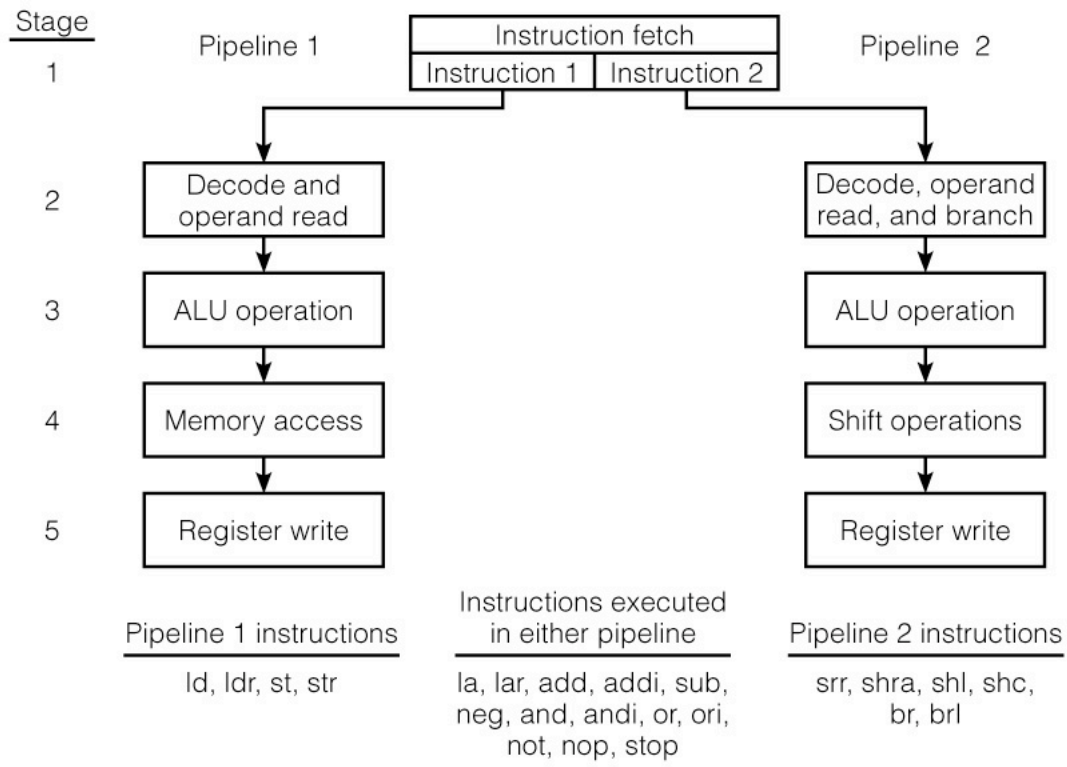 - Instructions at branch target may be prefetched, and even executed speculatively, in hopes the branch goes that way

Example 5.2: Dual Issue VLIW version of SRC

- Two instructions per word. Word size 2x32 (64) bits
- Two pipelines, each almost the same as the previous pipeline design.
- Only pipeline 1 can execute memory-access instructions: ld, ldr, st, and str
  - Thus only one memory access per cycle.
- Only pipeline 2 can execute shr shra shl shc, br, and brl
  - Assumes that a barrel shifter for the shift instructions is expensive and needed only in one pipeline, located in stage 4 replacing the memory access stage.
  - Limits the execution unit to one branch instruction per word.
- Either pipeline can execute the other instructions: la, lar, add, addi, sub, and, andi, or, ori, neg, not, nop, and stop.

Figure 5.16: Structure of the Dual-Pipeline SRC

Other features

- Register file may have 4 reads and two writes per cycle.
  - Either provide more read and write ports, or incorporate two register files, each an identical "shadow" copy of the other.
- No branch delay slot
- Instruction forwarding wherever possible.

Figures 5.17a and b: SRC Programs to Compute the Fibonacci Series on Single- and Dual-issue machines

(a) Program for a scalar version of SRC

```
; fib.asm. Compute Fibonacci numbers.
; The Fibonacci sequence is defined as follows:
; fib(1) = 1, fib(2) = 1,
; fib(n) = fib(n-1) + fib(n-2) n > 2.
cnt:    .equ 8              ; No. to compute after first two
        .org 0             ; Store sequence at addr. 0
seq:    .dc  1             ; Init. the first Fib. No.
next:   .dc  1             ; Init. the second Fib. No.
ans:    .dw  cnt           ; Storage for the next 8 Fib. Nos.
        .org 0x1000        ; Begin ass'y. at hex. addr. 1000
        lar  r31, loop     ; Branch address
        la   r0,  cnt      ; Init. count
        la   r1,  seq      ; Init r1 to index of seq[0]
loop: ld   r2,  seq(r1)    ; Get fib(n-2)
        ld   r3,  next(r1)  ; Get fib(n-1)
        add  r2,  r2,    r3 ; compute fib(n)
        st   r2,  ans(r1)   ; Store fib(n)
        addi r1,  r1,    4  ; Increment index
        addi r0,  r0,    -1 ; Decrement count
        brnz r31, r0        ; loop until done
        stop
```

(b) Program for a dual-issue version of SRC

| Pipeline 1	Pipeline 2	Line#
lar r31, loop	la r0, cnt	1
la r1, seq	nop	2
loop: ld r2, seq(r1)	addi r0, r0, -1	3
ld r3, next(r1)	nop	4
nop	nop	5
nop	add r2, r2, r3	6
st r2, ans(r1)	nop	7
addi r1, r1, 4	brnz r31, r0	8
stop	nop	9

© 2004 Prentice Hall

Fibonacci Program on the Dual-Issue Machine

- Total program length has been reduced from 11 lines to 9.
    - The loop, where the program will spend most of its time, has been reduced but only from 7 lines to 6 due to the imposed limitation that pipeline 2 cannot do memory accesses.
    - If this limitation were removed, both loads could take place at line 3.
    - The addi at line 3 would then be moved down to line 4, and line 5 could be eliminated.
- These loads still need to be separated by two from the sum of the two fibs in line 6 because of the hazard between load as writer and alu as reader. See Table 5.1.
- The store at line 7 needs to be separated by one from the add in line 6 because of the undefined semantics of computing a value and using it in an instruction in the same wide word.

Figure 5.19: Dual-Issue SRC Pipelines and Forwarding Paths
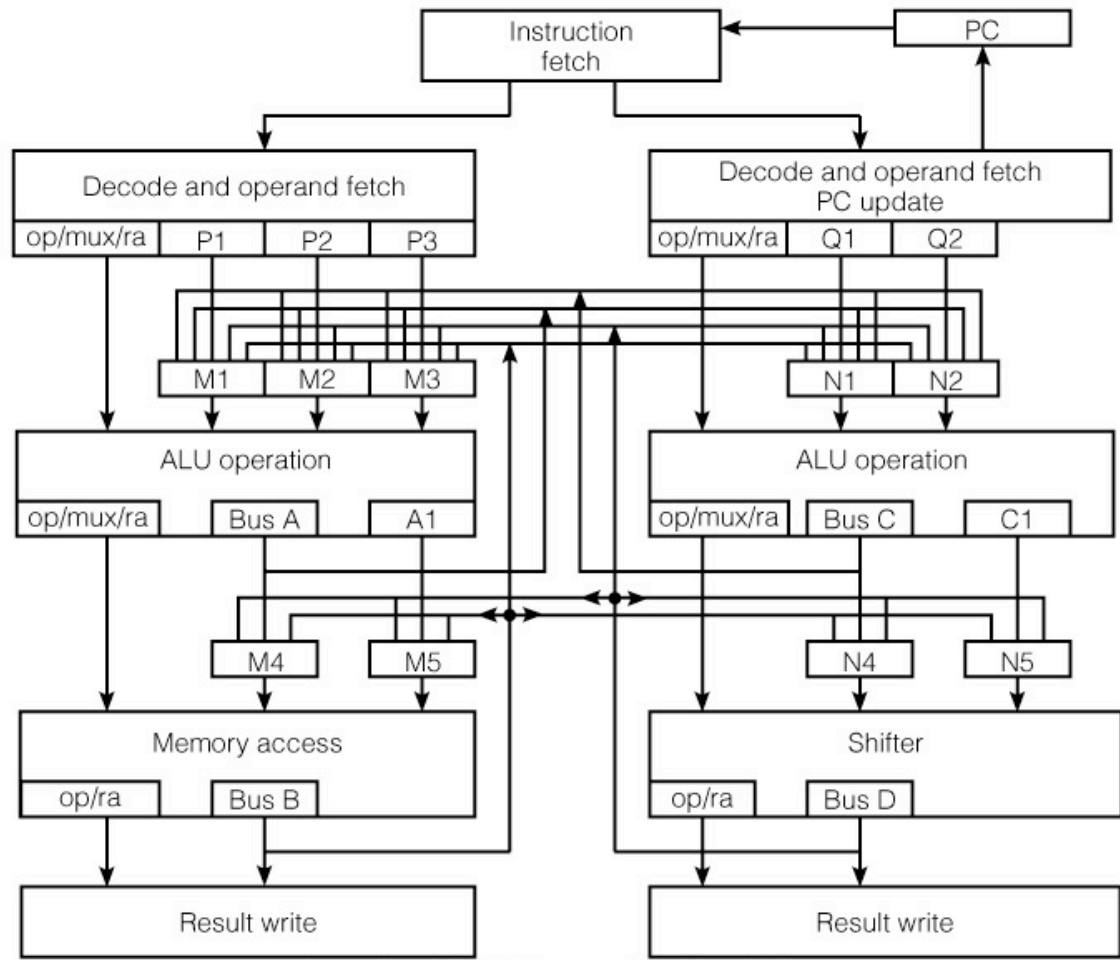
Copyright © 2004 Pearson Prentice Hall, Inc.

Figure 5.20: Dynamic Information in Dual-Issue SRC

(a) Instructions issued

Stage	Issue time	Instruction word	
		Instruction 1	Instruction 2
5	t − 2	add r6, r4, r4	shc r7, r1, 8
4	t − 1	addi r5, r3, 16	lar r8, 12
3	t	st r8, 4(r7)	sub r9, r5, r6

(b) Dynamic Table #1: Register status table for instruction in stage 3

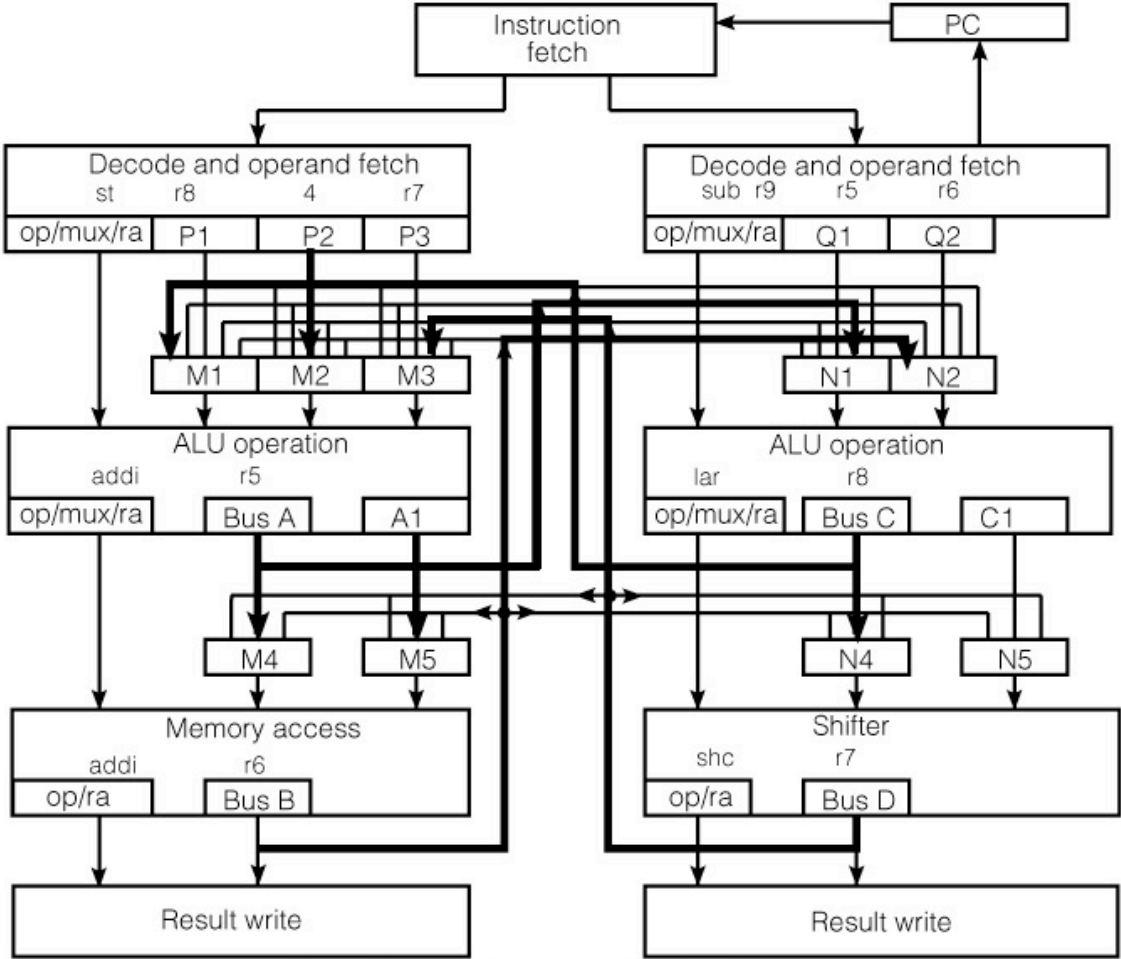| Register	Location†
R4	RF
R5	$(A, t);(B, t + 1)$
R6	$(B, t);(RF, t + 1)$
R7	$(D, t);(RF, t + 1)$
R8	$(C, t);(D, t + 1)$
R9	RF

† Bus A, B, C, D, or register file (RF)

(c) Dynamic Table #2: Op codes and multiplexer settings issued at time t

Pipeline 1	Pipeline 2
st M1=D, M2=P2, M3=C, M4=A, M5=A1	sub N1=A, N2=B, N4=C, N5=d

Figure 5.21: Operand Flow of st r8, 4(r7)

Getting Specific: Some Commercial Superscalar Processors

- PowerPC G4: Eleven pipelined functional units: 4 IUs, an FPU with a separate floating point register file, a BPU, an LSU, and 4 VPUs. It is capable of executing sixteen instructions simultaneously.

- Intel P6: Five functional units: 14-stage pipeline, 2 IUs, separate load and store units, FPU and BPU. Since the P6 must execute the CISC-like 80X86 instruction set, instructions entering the pipeline are decoded and fragmented into simpler RISC-like micro-ops, as they are called, which are dispatched to one of the five functional units. Instructions may be executed out of order, provided that doing so does not cause hazards.

- HP Alpha 21164: This processor has a 7-stage pipeline, 2 IUs, and 2 FPUs: one for add/subtract, and one for multiply/divide, branch prediction.
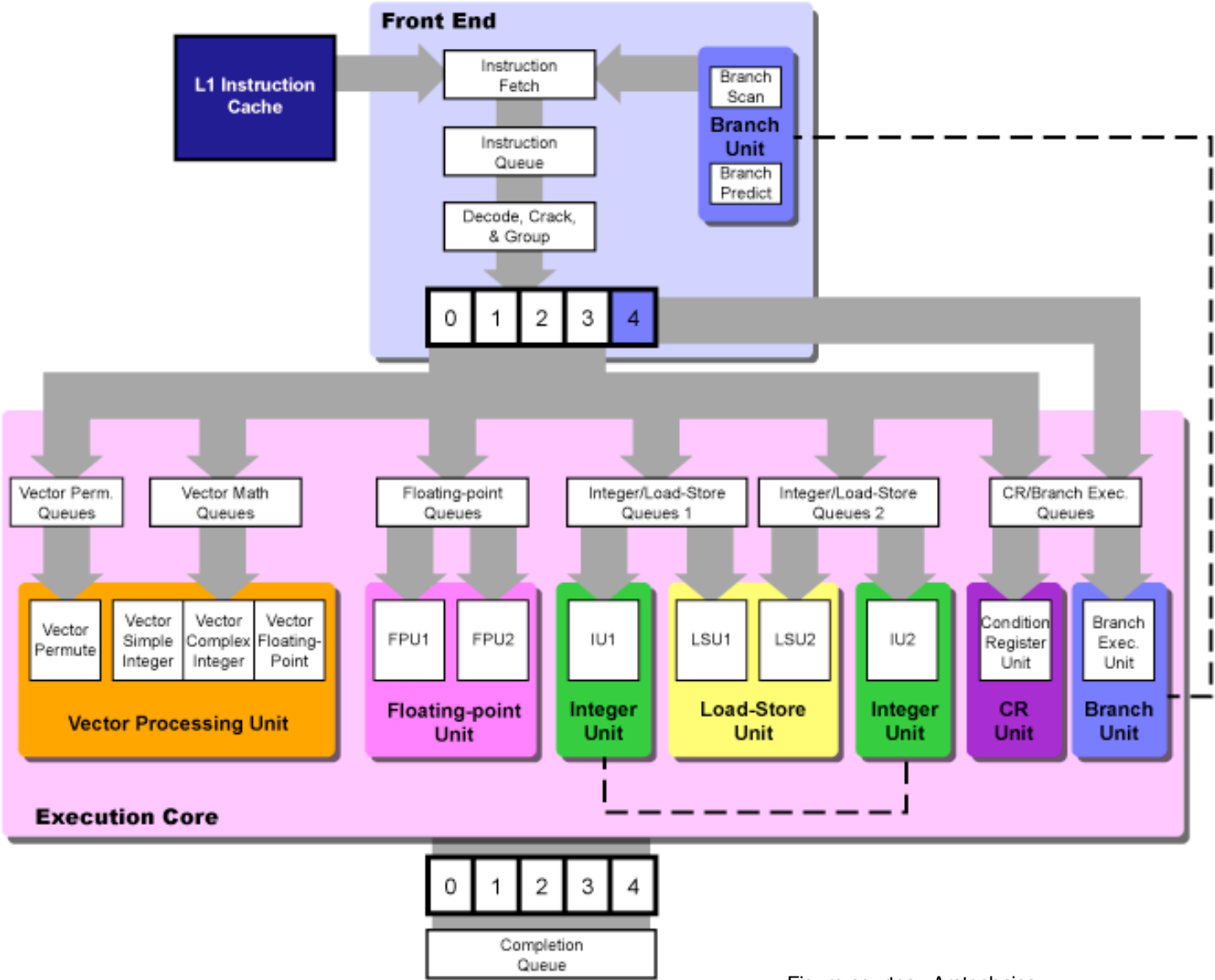
The Superscalar IBM PowerPC 970

Figure courtesy Arstechnica

Microprogramming: Basic Idea

- Recall control sequence for 1-bus SRC

Step	Concrete RTN	Control Sequence
T0.	MA ← PC: C ← PC+4;	PC_{out}, MA_{in}, Inc4, C_{in}, Read
T1.	MD ← M[MA]: PC ← C;	C_{out}, PC_{in}, Wait
T2.	IR ← MD;	MD_{out}, IR_{in}
T3.	A ← R[rb];	Grb, R_{out}, A_{in}
T4.	C ← A + R[rc];	Grc, R_{out}, ADD, C_{in}
T5.	R[ra] ← C;	C_{out}, Gra, R_{in}, End

- Control unit job is to generate the sequence of control signals
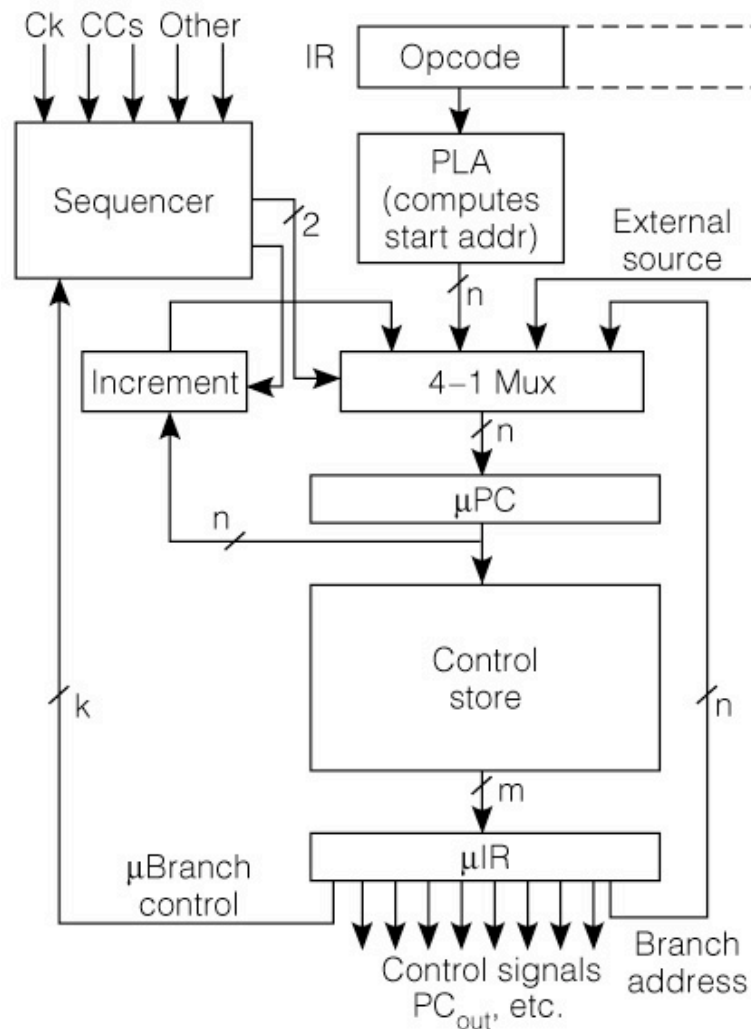- How about building a computer to do this?

The Microcode Engine

- A computer to generate control signals is <u>much</u> simpler than an ordinary computer

- At the simplest, it just reads the control signals in order from a read only memory

- The memory is called the <u>control store</u>

- A control store word, or <u>microinstruction</u>, contains a bit pattern telling which control signals are true in a specific step

- The major issue is determining the order in which microinstructions are read

Fig 5.22 Block Diagram of a Microcoded Control Unit

- Microinstruction has branch control, branch address, and control signal fields
- Micro-program counter can be set from several sources to do the required sequencing

Ck CCs Other

IR Opcode

Sequencer

2

PLA (computes start addr)

External source

n

Increment

4–1 Mux

n

n

μPC

k

Control store

n

m

μBranch control

μIR

Control signals PC_out, etc.

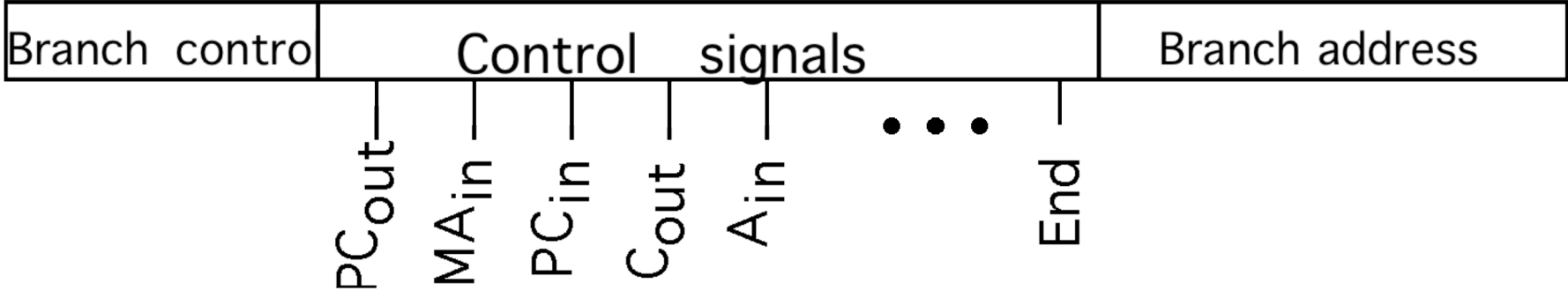Branch address

Copyright © 2004 Pearson Prentice Hall, Inc.

Parts of the Microprogrammed Control Unit

- Since the control signals are just read from memory, the main function is sequencing

- This is reflected in the several ways the μPC can be loaded

  - Output of incrementer—μPC+1

  - PLA output—start address for a macroinstruction

  - Branch address from μinstruction

  - External source—say for exception or reset

- Micro conditional branches can depend on condition codes, data path state, external signals, etc.
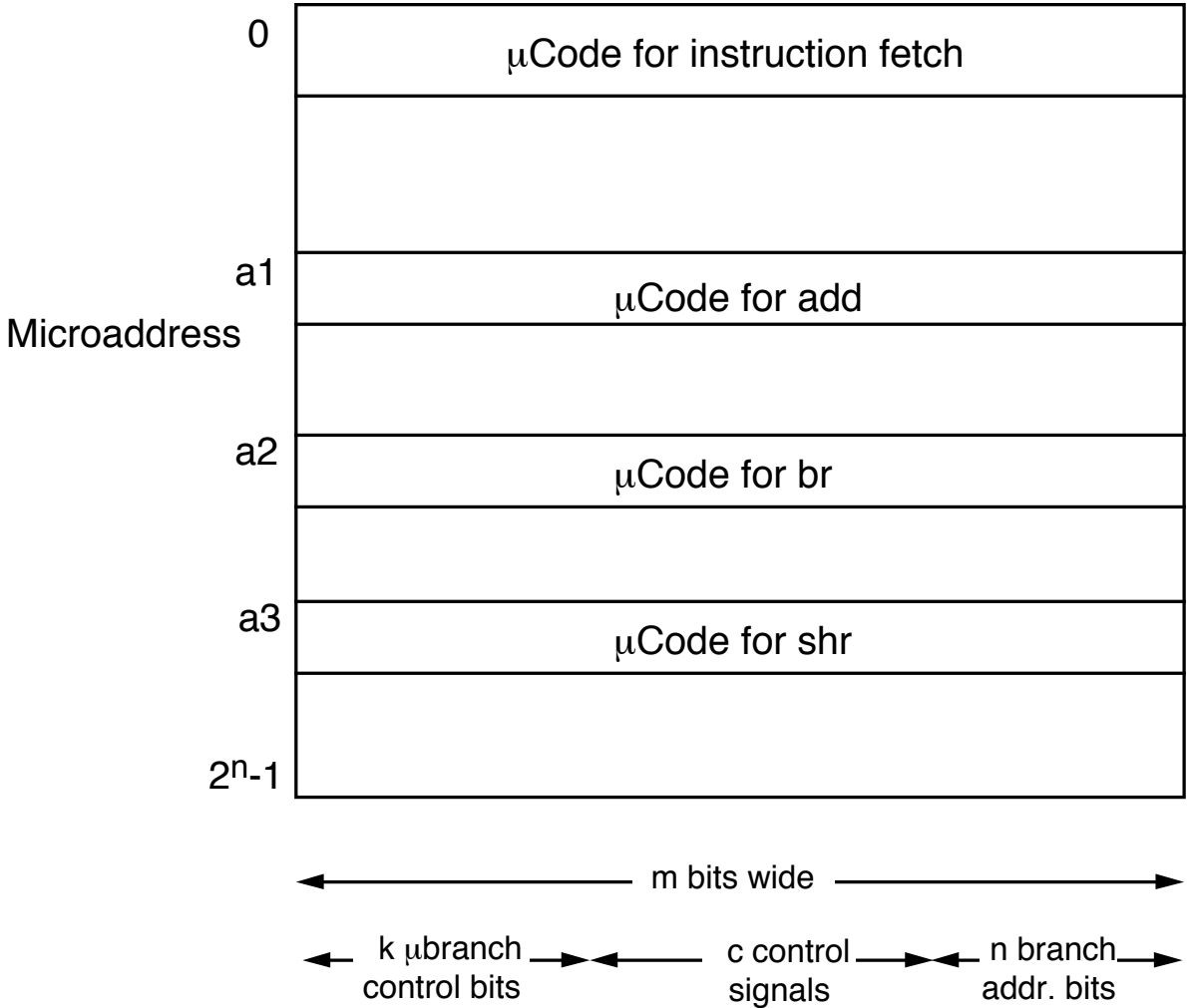
Contents of a Microinstruction

Microinstruction format

Branch contro	Control signals	Branch address

PC$_{out}$ MA$_{in}$ PC$_{in}$ C$_{out}$ A$_{in}$ • • •   End

- Main component is list of 1/0 control signal values
- There is a branch address in the control store
- There are branch control bits to determine when to use the branch address and when to use μPC+1

Figure 5.23: Layout of the Control Store

0	μCode for instruction fetch
a1	μCode for add
a2	μCode for br
a3	μCode for shr
2^n-1	

Microaddress

- Common inst. fetch sequence
- Separate sequences for each (macro) instruction
- Wide words

m bits wide

k μbranch control bits | c control signals | n branch addr. bits

Size and Shape of System RAM vs Control Store

- System RAM is one byte wide x 2^{32} bytes deep.

- Assume control store has 128 instructions, 128 bits wide, with 8 steps each.

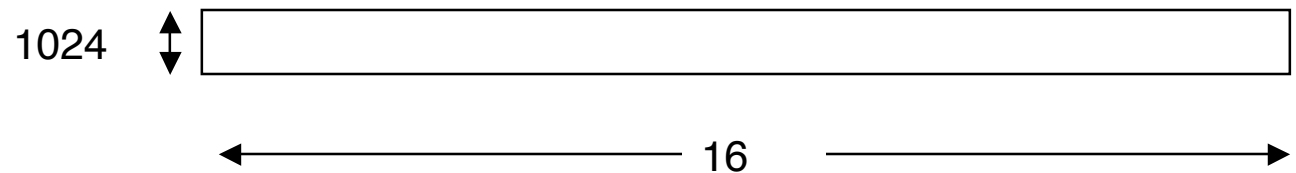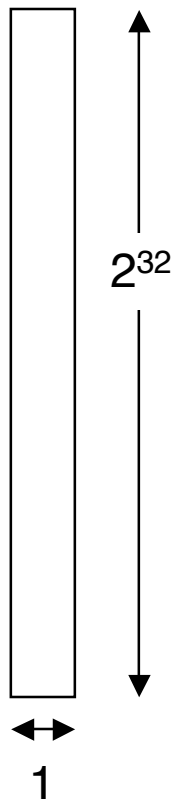- Control store would be 16 bytes wide, but only 128x8 or 1024 words deep.

2^{32}

1

1024

16

Table 5.2: Microinstruction Control Signals for the add Instruction

Address	Branch Control	PCout	Cout	MDout	Rout	MAin	Cin	PCin	IRin	Ain	Rin	Inc4	Read	Wait	ADD	Gra	Grb	Grc	End
101	· · ·	1	0	0	0	1	1	0	0	0	0	1	1	0	0	0	0	0	0
102	· · ·	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0
103	· · ·	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
200	· · ·	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0
201	· · ·	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	1	0
202	· · ·	0	1	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	1

- Addresses 101–103 are the instruction fetch
- Addresses 200–202 do the add
- Change of μcontrol from 103 to 200 uses a kind of μbranch

Uses for μbranching in the Microprogrammed Control Unit

- 1) Branch to start of μcode for a specific inst.

- 2) Conditional control signals, e.g. CON $\rightarrow$ PC_{in}

- 3) Looping on conditions, e.g. $n \neq 0$ $\rightarrow$ ... Goto6

- Conditions will control μbranches instead of being ANDed with control signals

- Microbranches are frequent and control store addresses are short, so it is reasonable to have a μbranch address field in every μ instruction

Illustration of μbranching Control Logic

- We illustrate a μbranching control scheme by a machine having condition code bits N & Z

- Branch control has 2 parts:

- 1) selecting the input applied to the μPC and

- 2) specifying whether this input or μPC+1 is used

- We allow 4 possible inputs to μPC

  - The incremented value μPC+1

  - The PLA lookup table for the start of a macroinstruction

  - An externally supplied address

  - The branch address field in the μinstruction word

Fig 5.24 Branching Controls in the Microcoded Control Unit
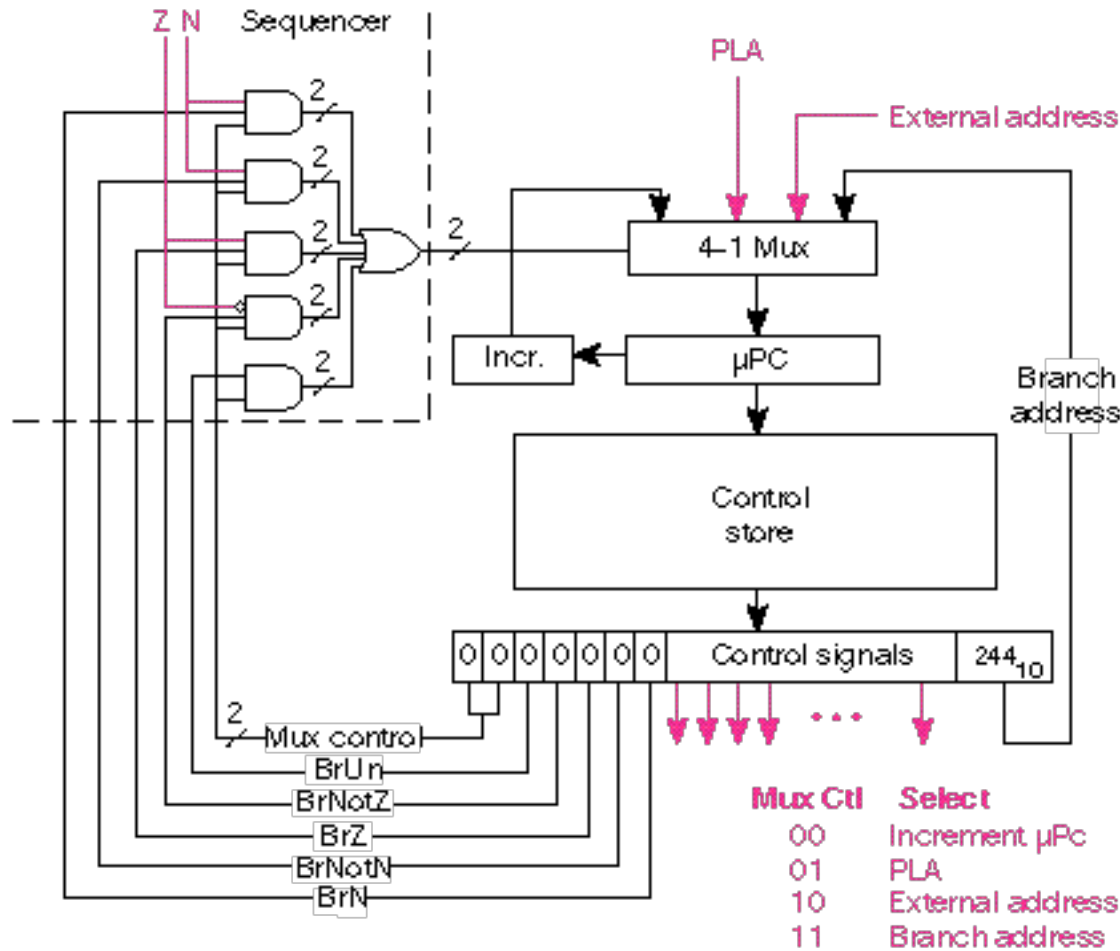
- 5 branch conditions
  - NotN
  - N
  - NotZ
  - Z
  - Uncondit.
- To 1 of 4 places
  - Next μinst.
  - PLA
  - Extern. addr.
  - Branch addr.

Some Possible μbranches Using the Illustrated Logic

| Mux Ctl	BrUn	BrNotZ	BrZ	BrNotN	BrN	Control Signals	Branch Address	Branching action
00	0	0	0	0	0	•••	XXX	None—next instruction
01	1	0	0	0	0	•••	XXX	Branch to output of PLA
10	0	0	1	0	0	•••	XXX	Br if Z to Extern. Addr.
11	0	0	0	0	1	•••	300	Br if N to 300 (else next)
11	0	0	0	1	0	0•••0	206	Br if N̄ to 206 (else next)
11	1	0	0	0	0	•••	204	Br to 204

- If the control signals are all zero, the μinst. only does a test
- Otherwise test is combined with data path activity

# Horizontal Versus Vertical Microcode Schemes

- In <u>horizontal</u> microcode, each control signal is represented by a bit in the μinstruction

- In <u>vertical</u> microcode, a set of true control signals is represented by a shorter code

- The name horizontal implies fewer control store words of more bits per word

- Vertical μcode only allows RTs in a step for which there is a vertical μinstruction code

- Thus vertical μcode may take more control store words of fewer bits

Fig 5.25 A Somewhat Vertical Encoding
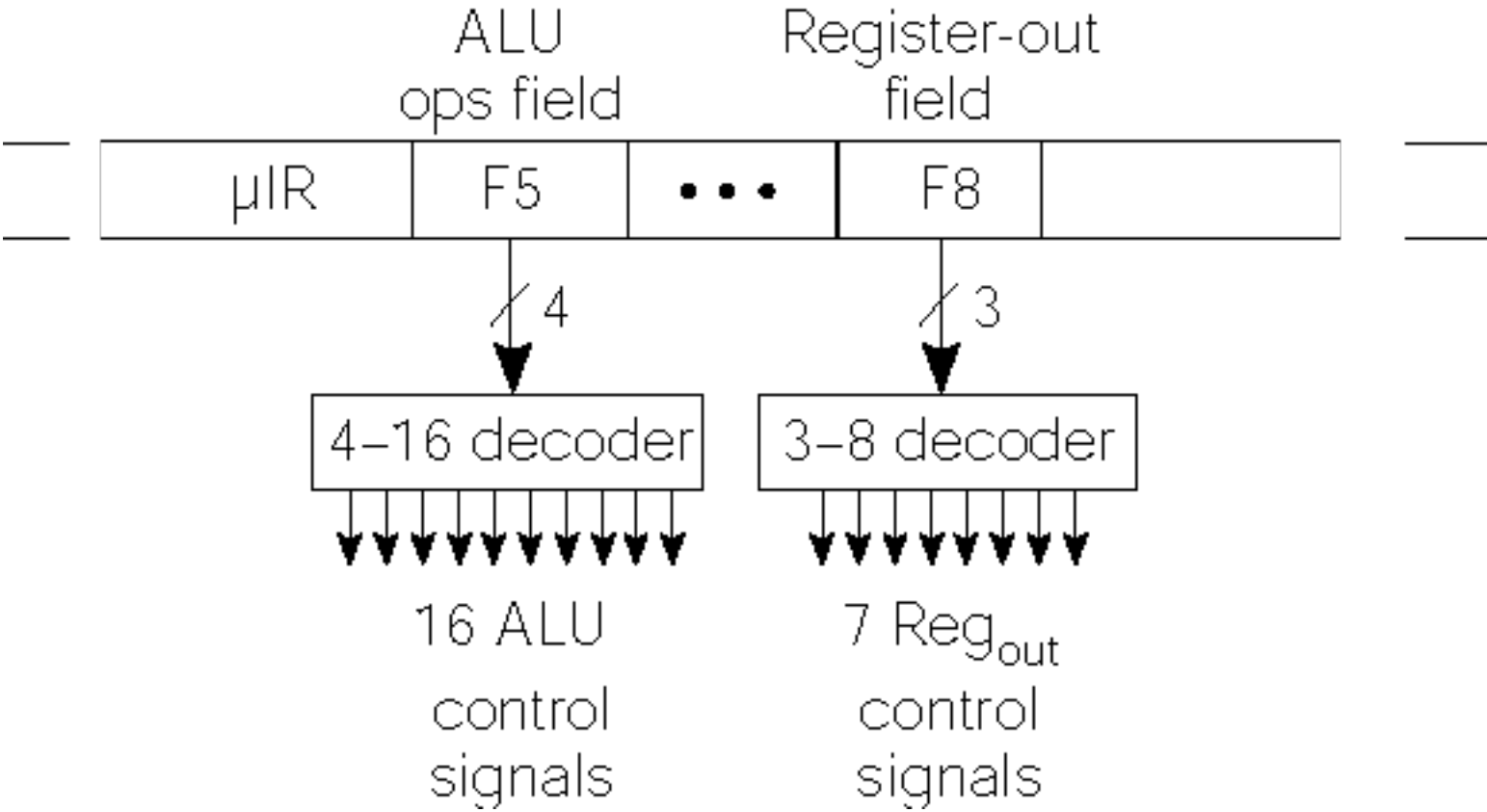
- Scheme would save (16+7) - (4+3) = 16 bits/word in the case illustrated

Fig 5.26 Completely Horizontal and Vertical Microcoding
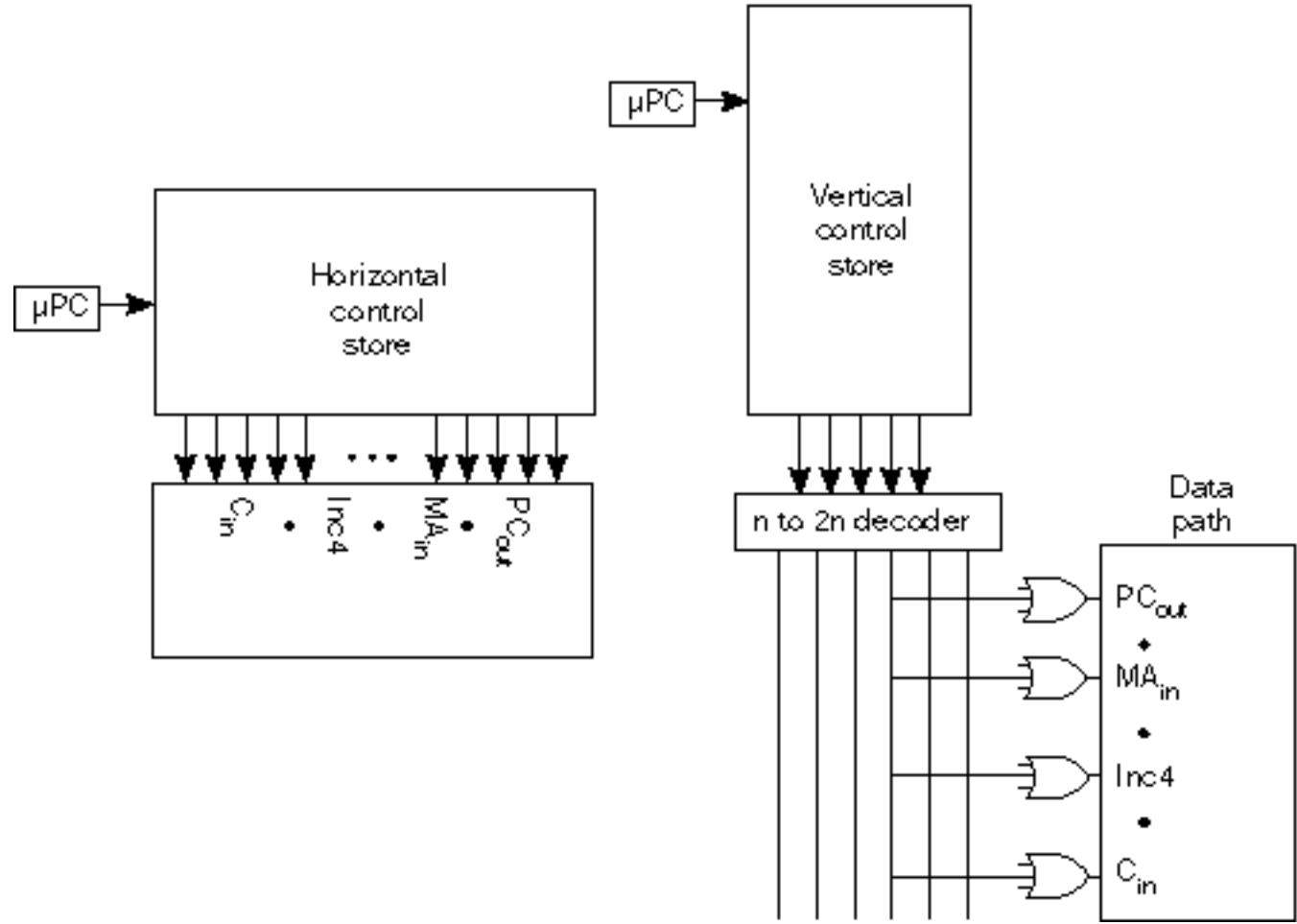
Saving Control Store Bits With Horizontal Microcode

- Some control signals cannot possibly be true at the same time
  - One and only one ALU function can be selected
  - Only one register out gate can be true with a single bus
  - Memory read and write cannot be true at the same step
- A set of m such signals can be encoded using $\log_2 m$ bits ($\log_2(m+1)$ to allow for no signal true)
- The raw control signals can then be generated by a k to 2^k decoder, where $2^k \geq m$ (or $2^k \geq m+1$)
- This is a compromise between horizontal and vertical encoding

A Microprogrammed Control Unit for the 1-bus SRC

- Using the 1-bus SRC data path design gives a specific set of control signals

- There are no condition codes, but data path signals CON and n=0 will need to be tested

- We will use μbranches BrCON, Brn=0, & Brn≠0

- We adopt the clocking logic of Fig. 4.9 on p. 4-20

- Logic for exception and reset signals is added to the microcode sequencer logic

- Exception and reset are assumed to have been synchronized to the clock

Table 5.4 Microinstructions for SRC add

Addr.	Mux Ctl	BrUn	BrCON	Brn≠0	Brn=0	End	PC_{out}	MA_{in}	Other Control Signals	Br Addr.	Actions
100	00	0	0	0	0	0	1	1	• • •	XXX	MA ← PC: C ← PC+4;
101	00	0	0	0	0	0	0	0	• • •	XXX	MD ← M[MA]: PC ← C;
102	01	1	0	0	0	0	0	0	• • •	XXX	IR ← MD; μPC ← PLA;
200	00	0	0	0	0	0	0	0	• • •	XXX	A ← R[rb];
201	00	0	0	0	0	0	0	0	• • •	XXX	C ← A + R[rc];
202	11	1	0	0	0	1	0	0	• • •	100	R[ra] ← C: μPC ← 100;

- Microbranching to the output of the PLA is shown at 102
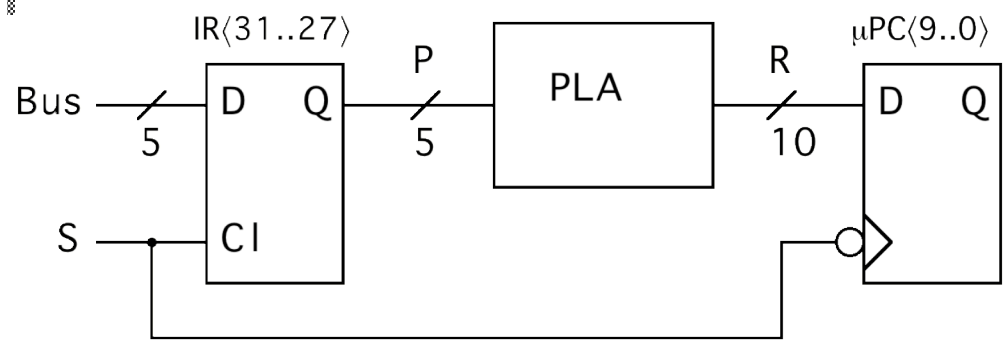- Microbranch to 100 at 202 starts next fetch

Getting the PLA Output in Time for the Microbranch

- So that the input to the PLA is correct for the μbranch in 102, it has to come from MD, not IR

- An alternative is to use see-thru latches for IR so the op code can pass through IR to PLA before the end of the clock cycle

See-thru Latch Hardware for IR So μPC Can Load Immediately

- Data must have time to get from MD across Bus, through IR, through the PLA, and satisfy μPC set up time before trailing edge of S
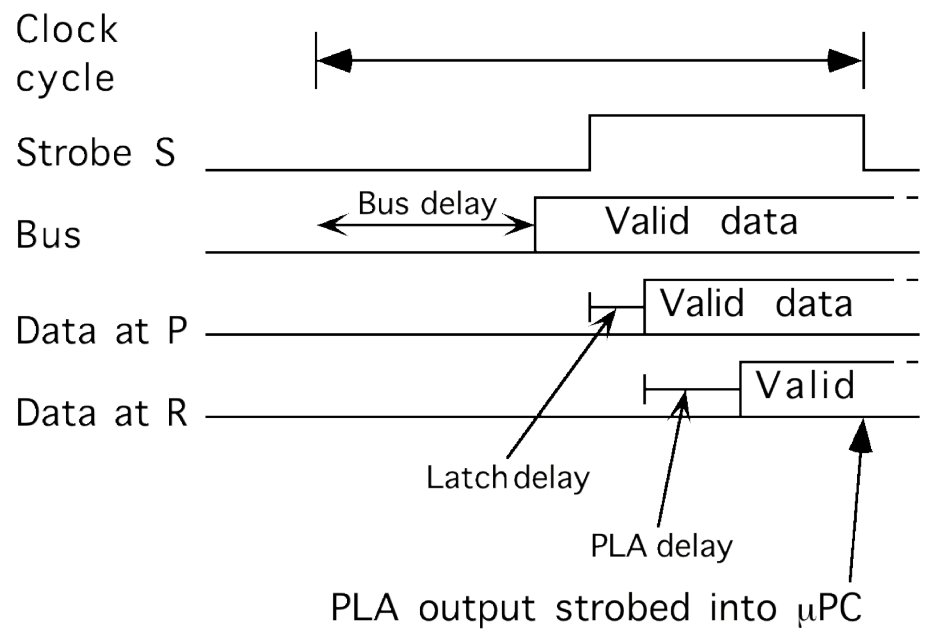
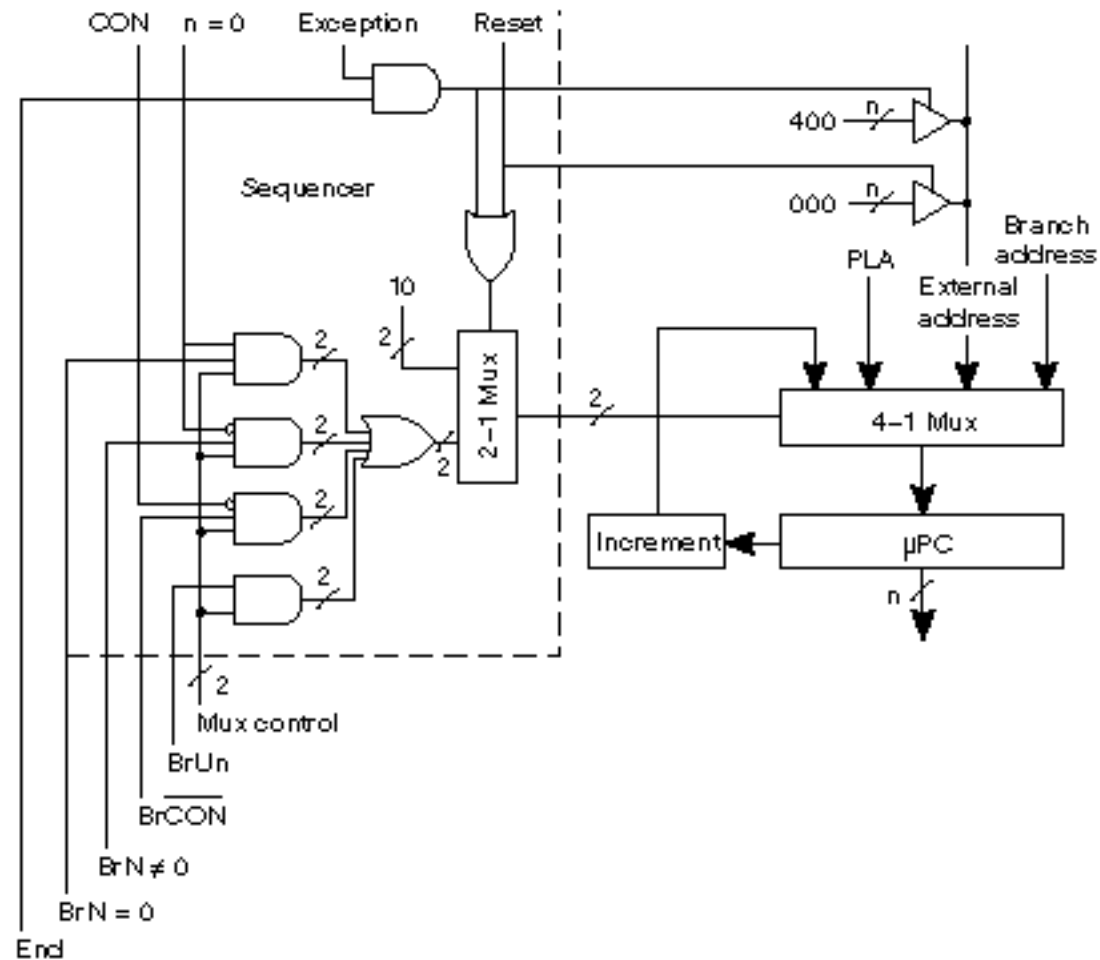# Fig 5.27 Microcode Sequencer Logic for SRC

Table 5.6 A Somewhat Vertical Encoding of the SRC Microinstruction

F 1	F 2	F 3	F 4	F 5	F 6	F 7	F 8	F 9
Mux Ctl	Branch control	End	Out signals	In signals	Misc.	Gate regs.	ALU	Branch address
00 01 10 11	000 BrUn 001 Br¬CON 010 BrCON 011Br n=0 100 Br n≠0 101 None	0 Cont. 1 End	000 PC_{out} 001 C_{out} 010 MD_{out} 011 R_{out} 100 BA_{out} 101 $c1_{out}$ 110 $c2_{out}$ 111 None	000 MA_{in} 001 PC_{in} 010 IR_{in} 011 A_{in} 100 R_{in} 101 MD_{in} 110 None	000 Read 001 Wait 010 Ld 011 Decr 100 CON_{in} 101 C_{in} 110 Stop 111 None	00 Gra 01 Grb 10 Grc 11 None	0000 ADD 0001 C=B 0010 SHR 0011 Inc4 • • • 1111 NOT	10 bits
2bits	3 bits	1 bit	3 bits	3 bits	3 bits	2 bits	4 bits	10 bits

Other Microprogramming Issues

- Multi-way branches: often an instruction can have 4-8 cases, say address modes
  - Could take 2-3 successive μbranches, i.e. clock pulses
  - The bits selecting the case can be ORed into the branch address of the μinstruction to get a several way branch
  - Say if 2 bits were ORed into the 3rd & 4th bits from the low end, 4 possible addresses ending in 0000, 0100, 1000, and 1100 would be generated as branch targets
  - Advantage is a multi-way branch in one clock
- A hardware push-down stack for the μPC can turn repeated μsequences into μsubroutines
- Vertical μcode can be implemented using a horizontal μengine, sometimes called nanocode

Chapter 5 Summary

- This chapter has dealt with some alternative ways of designing a computer

- A pipelined design is aimed at making the computer fast—target of one inst. per clock

- Forwarding, branch delay slot, and load delay slot are steps in approaching this goal

- A static multiissue SRC design shows some of the strengths and limitations of this architecture.

- Microprogramming is a design method with a target of easing the design task, and allowing for easy design change or multiple compatible implementations of the same instruction set