

Chapter 5: Processor Design— Advanced Topics

Topics

5.1 Pipelining

- A pipelined design of SRC
- Pipeline hazards

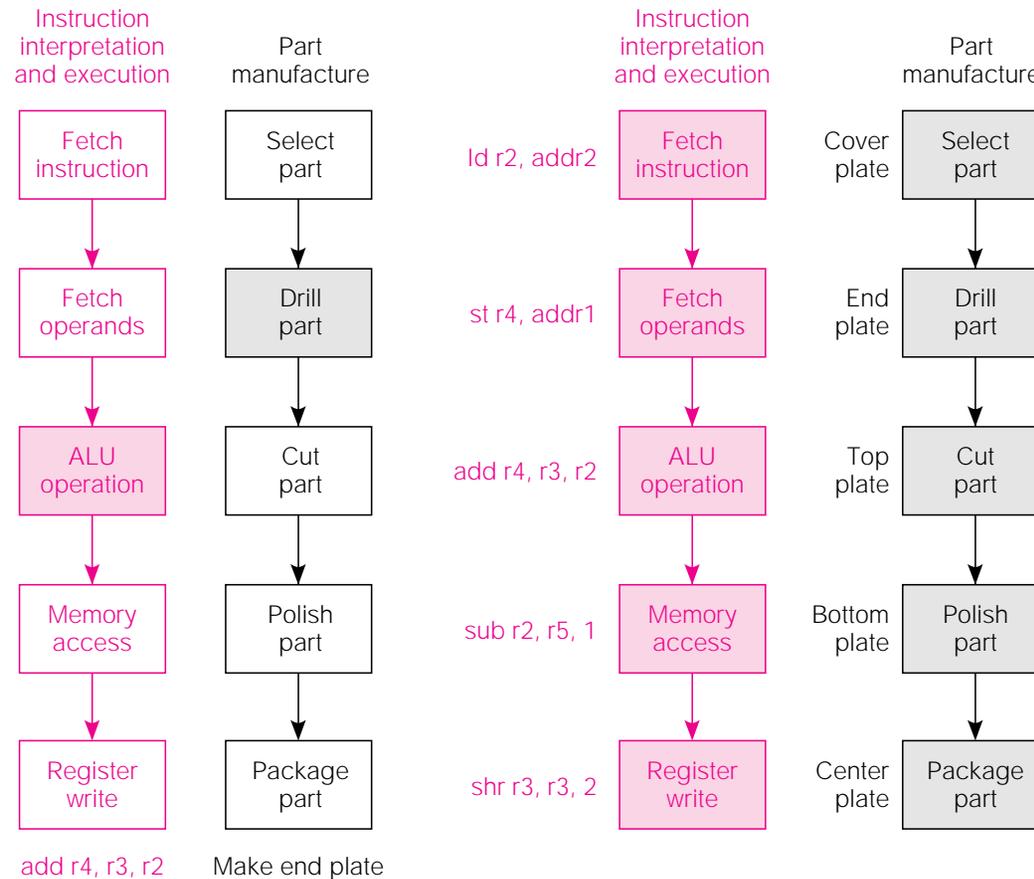
5.2 Instruction-Level Parallelism

- Superscalar processors
- Very Long Instruction Word (VLIW) machines

5.3 Microprogramming

- Control store and microbranching
- Horizontal and vertical microprogramming

Fig 5.1 Executing Machine Instructions versus Manufacturing Small Parts



(a) Without pipelining/assembly line

(b) With pipelining/assembly line

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 into r3`
 - `sub r2, r5, #1 ;Idle—no memory access needed`
 - `add r4, r3, r2 ;Performing addition in ALU`
 - `st r4, addr1 ;Accessing r4 and addr1`
 - `ld r2, addr2 ;Fetching instruction`

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 instructions in pipeline is reverse of listed
- If each stage takes 1 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  
st r6, addr1
```

← This instruction always executed

← Only done if $r2 \neq 0$

Load Delay

```
ld r2, addr  
add r5, r1, r2  
shr r1, r1, #4  
sub r6, r8, r2
```

← This instruction gets “old” value of r2

← This instruction gets r2 value loaded from addr

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

Characteristics of Pipelined Processor Design

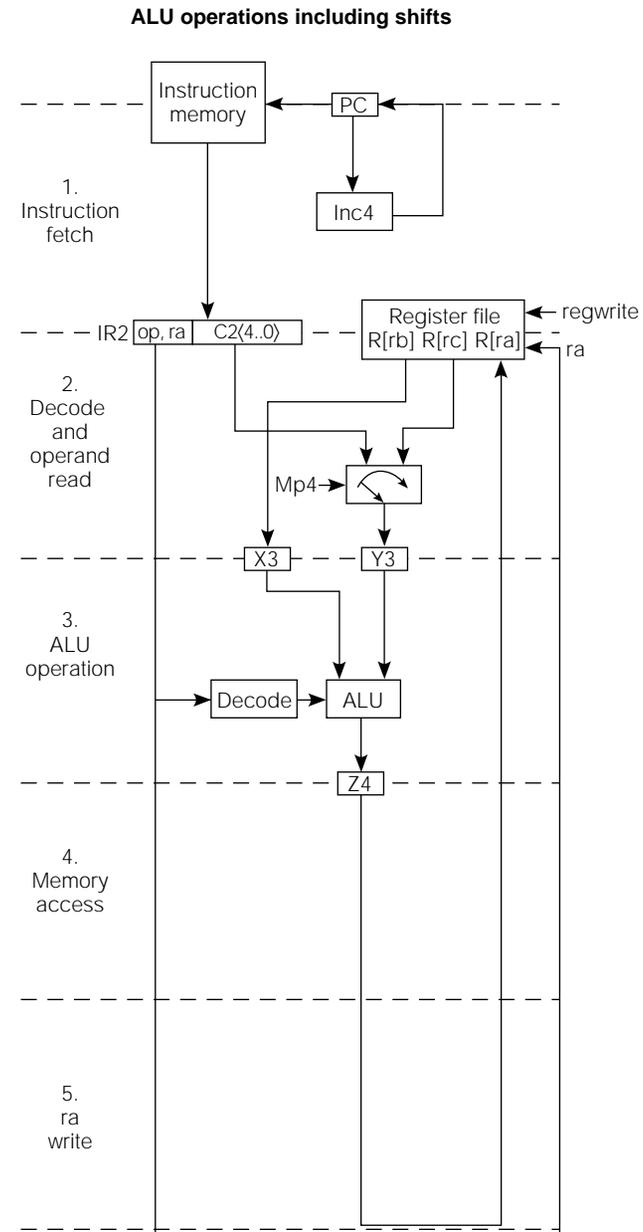
- **Main memory must operate in one cycle**
 - This can be accomplished by expensive memory, but
 - It is usually done with cache, to be discussed in Chap. 7
- **Instruction and data memory must appear separate**
 - Harvard architecture has separate instruction and data memories
 - Again, this is usually 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

- Instructions fit into 5 stages
- Second ALU operand comes either from a register or instruction register c2 field
- Opcode must be available in stage 3 to tell ALU what to do
- Result register, ra, is written in stage 5
- No memory operation



Logic Expressions Defining Pipeline Stage Activity

branch := br \vee brl :

cond := $(\text{IR2}\langle 2..0 \rangle = 1) \vee ((\text{IR2}\langle 2..1 \rangle = 1) \wedge (\text{IR2}\langle 0 \rangle \oplus \text{R}[\text{rb}] = 0)) \vee$
 $((\text{IR2}\langle 2..1 \rangle = 2) \wedge \overline{(\text{IR2}\langle 0 \rangle \oplus \text{R}[\text{rb}] \langle 31 \rangle)}) :$

sh := shr \vee shra \vee shl \vee shc :

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

imm := addi \vee andi \vee ori \vee (sh \wedge (IR2 $\langle 4..0 \rangle \neq 0$)) :

load := ld \vee ldr :

ladr := la \vee lar :

store := st \vee str :

l-s := load \vee ladr \vee store :

regwrite := load \vee ladr \vee brl \vee alu: Instructions that write to the register file

dsp := ld \vee st \vee lar : Instructions that use disp addressing

rl := ldr \vee str \vee 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 $\text{load5} \vee \text{ladr5} \vee \text{brl5} \vee \text{alu5}$, all of which are determined from op5

Fig 5.4 The Memory Access Instructions: Id, Idr, st, and str

- ALU computes effective addresses
- Stage 4 does read or write
- Result register written only on load

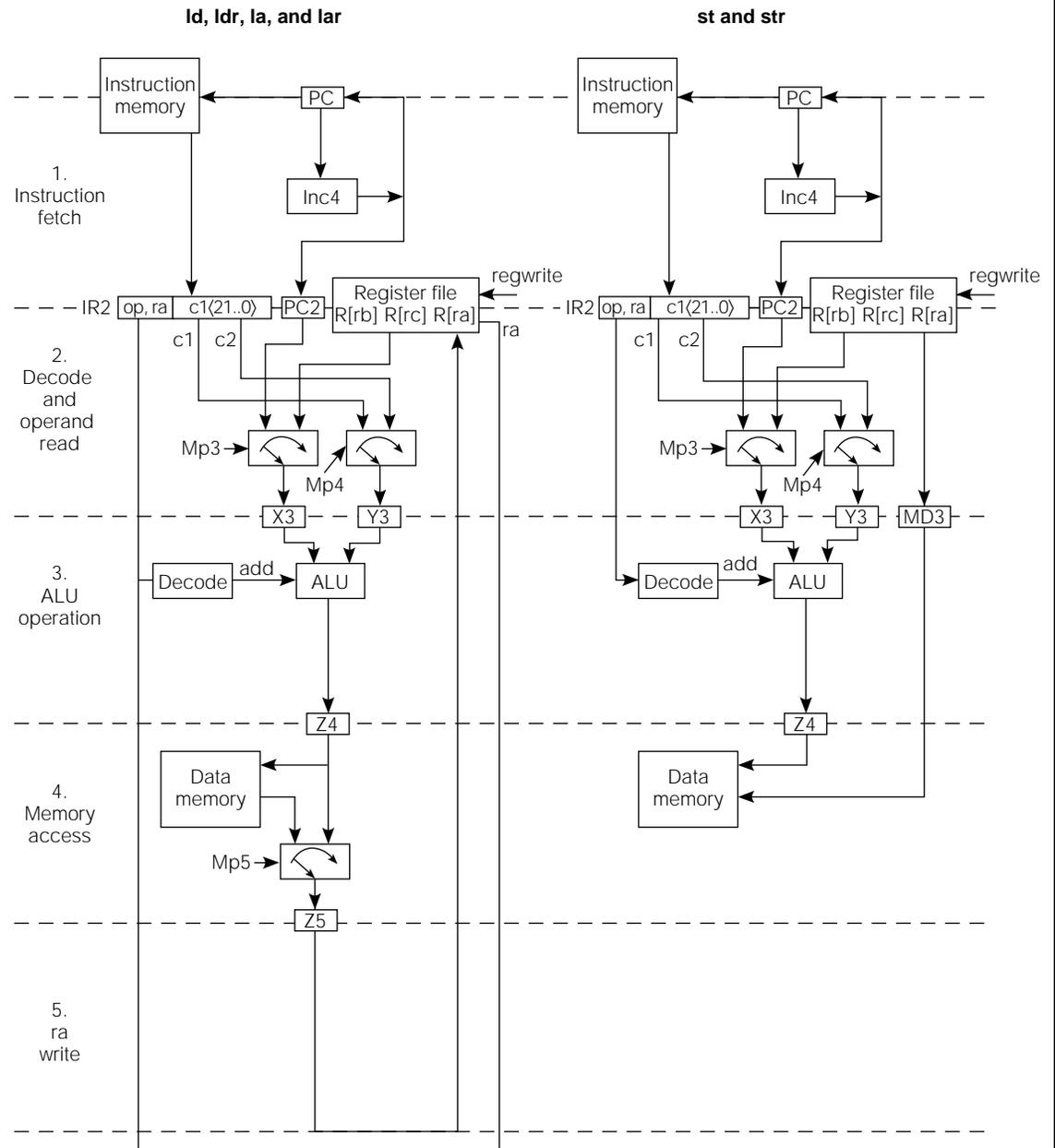
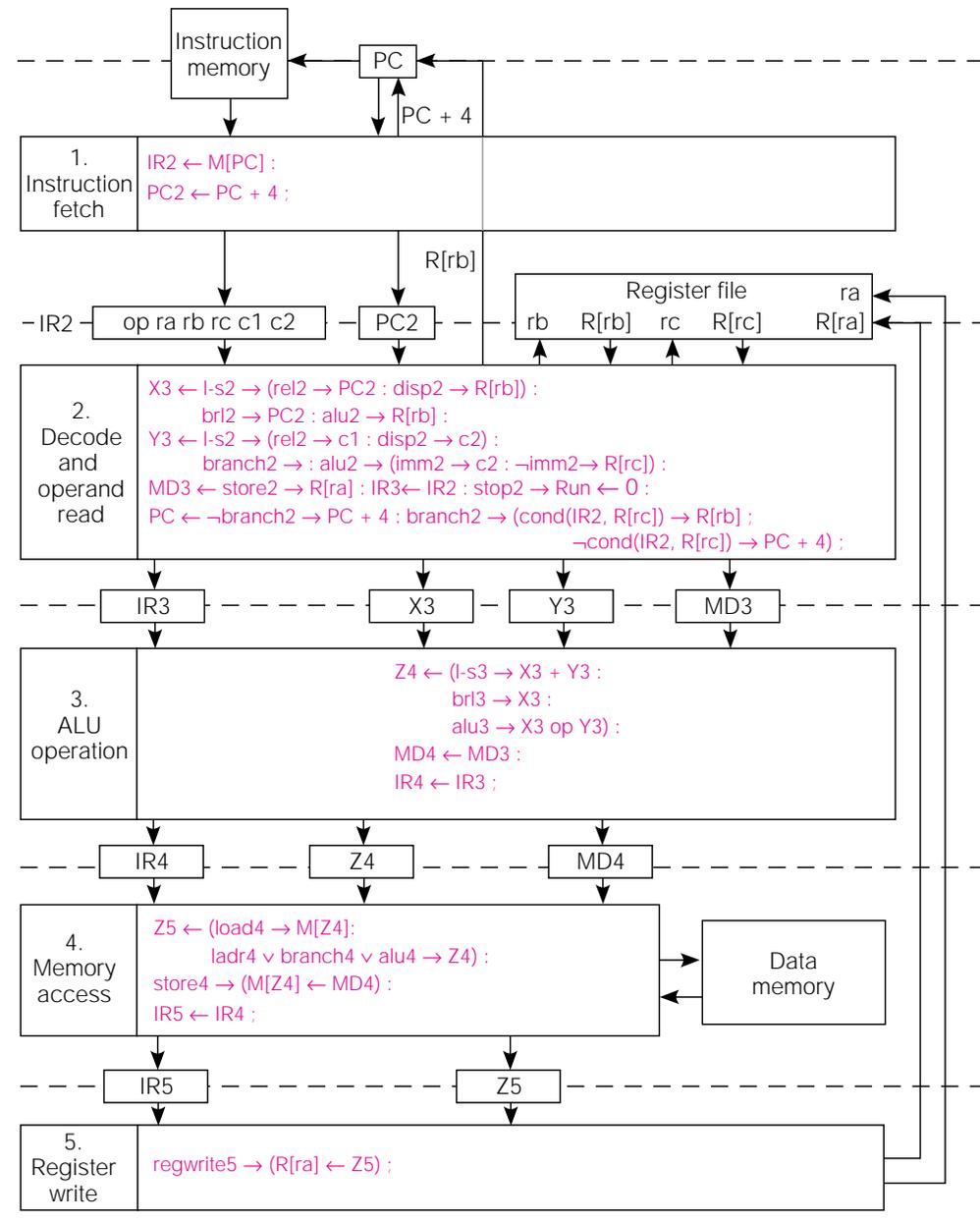


Fig 5.6 The SRC Pipeline Registers and RTN Specification

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



Global State of the Pipelined SRC

- PC, the general registers, instruction memory, and data memory represent the global machine state
- PC is accessed in stage 1 (and 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

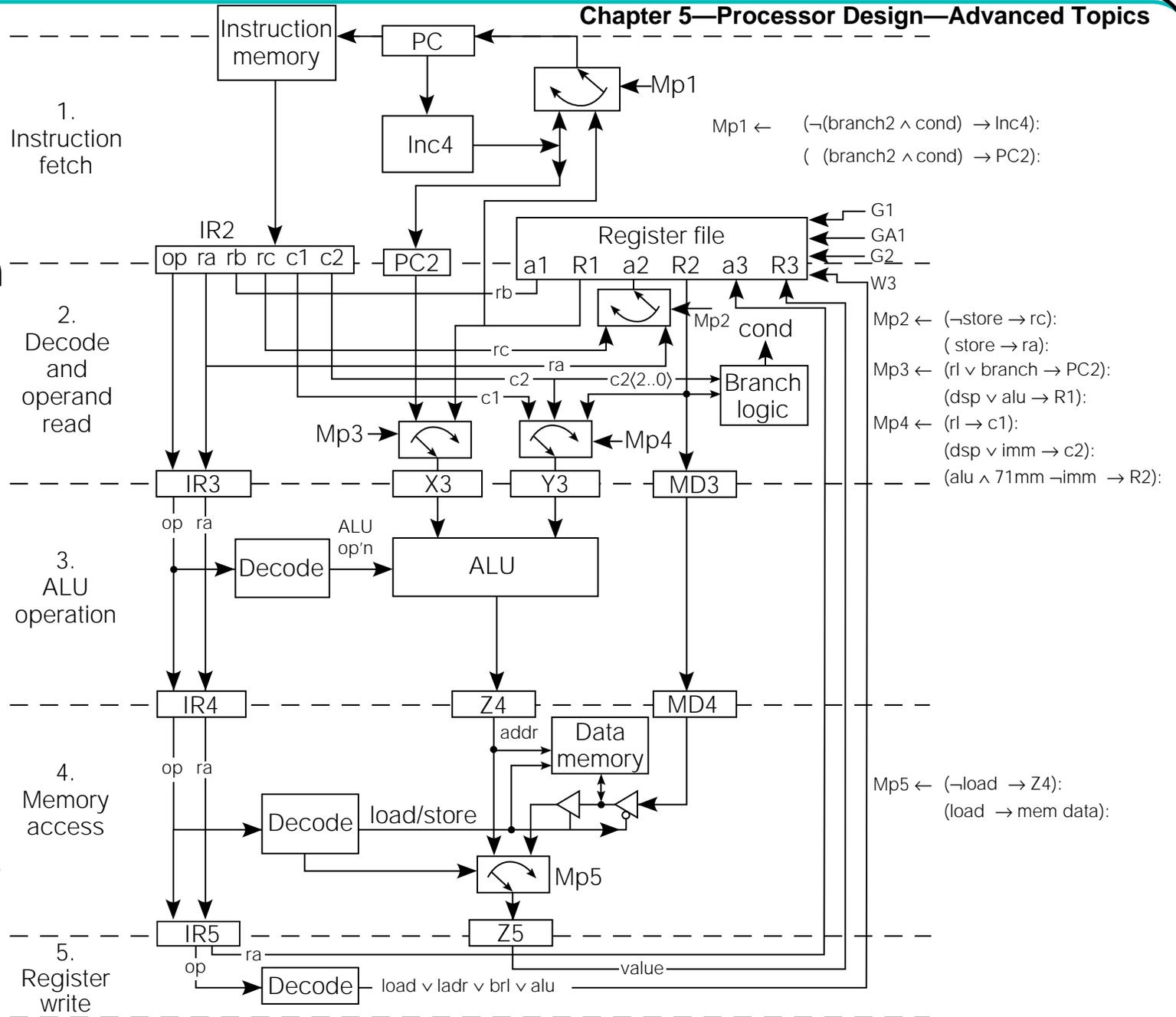
Restrictions on Access to Global State by Pipeline

- 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

5-17

Fig 5.7 The Pipeline Data Path with Selected Control Signals

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



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 instr. ...

```

- 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 First Clock Cycle: add Enters Stage 1 of Pipeline

- 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

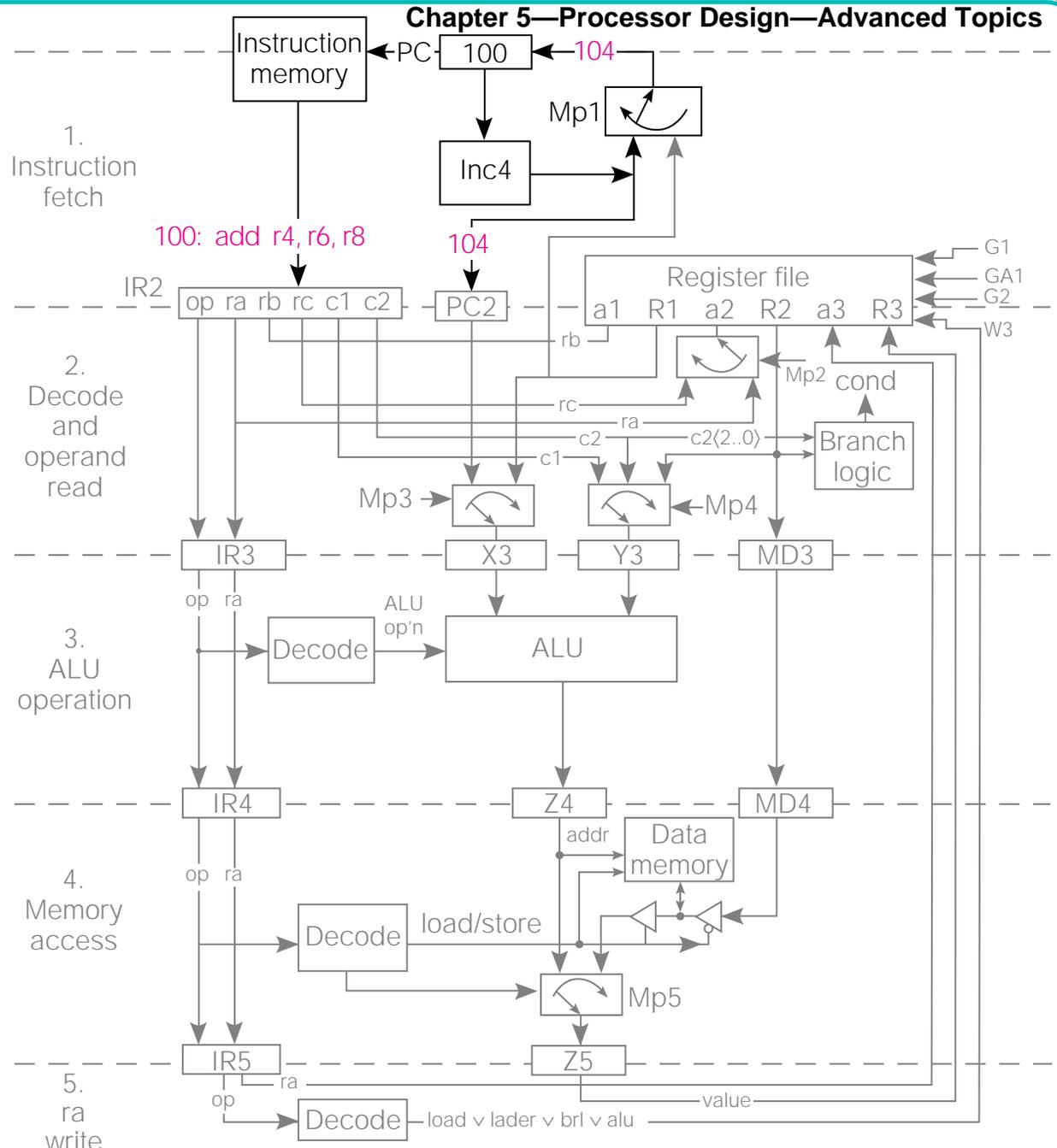


Fig 5.9 Second Clock Cycle: add Enters Stage 2, While 1d is Being Fetched at Stage 1

- 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

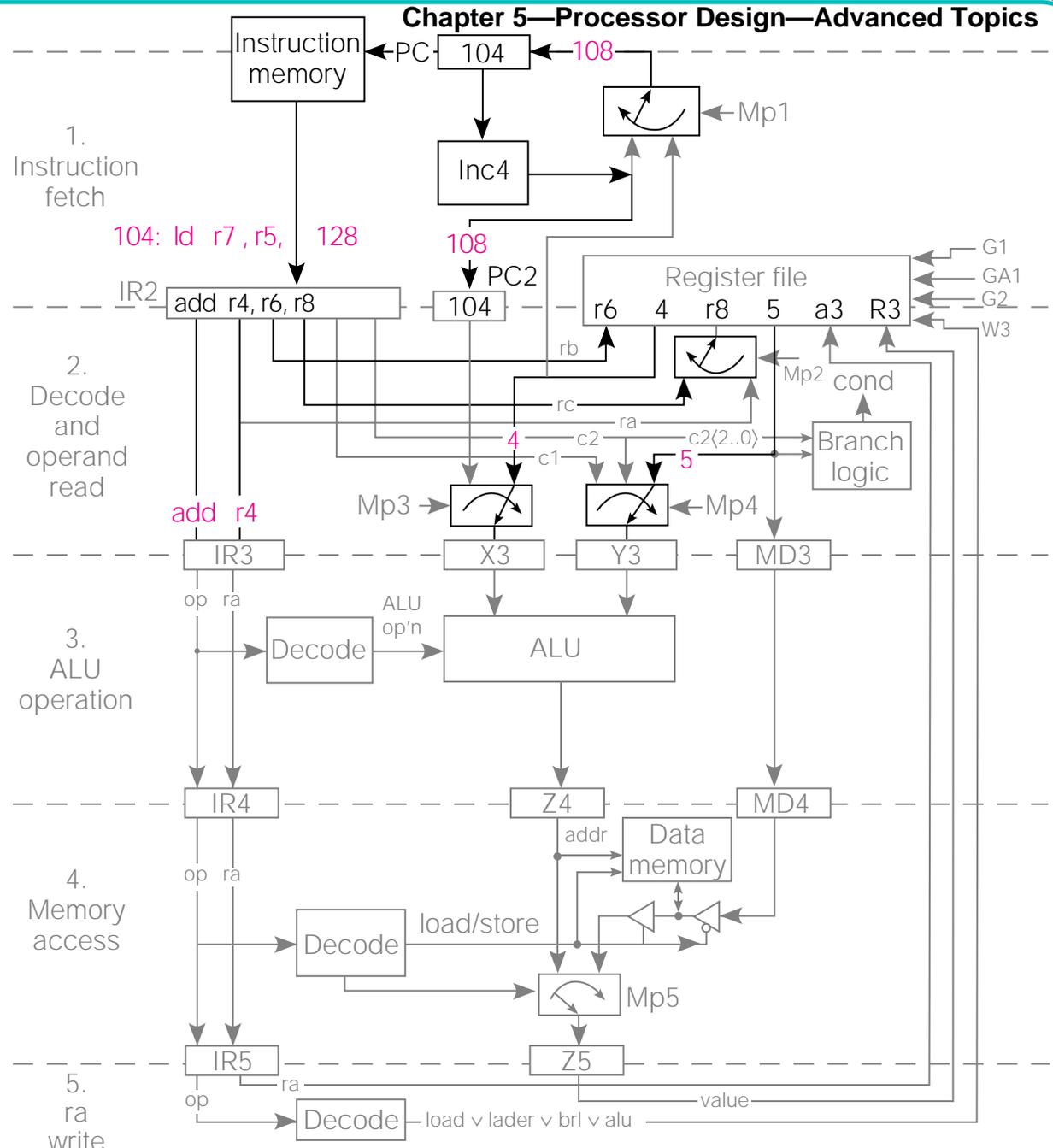


Fig 5.11 Fourth Clock Cycle: str Enters the Pipeline

- 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

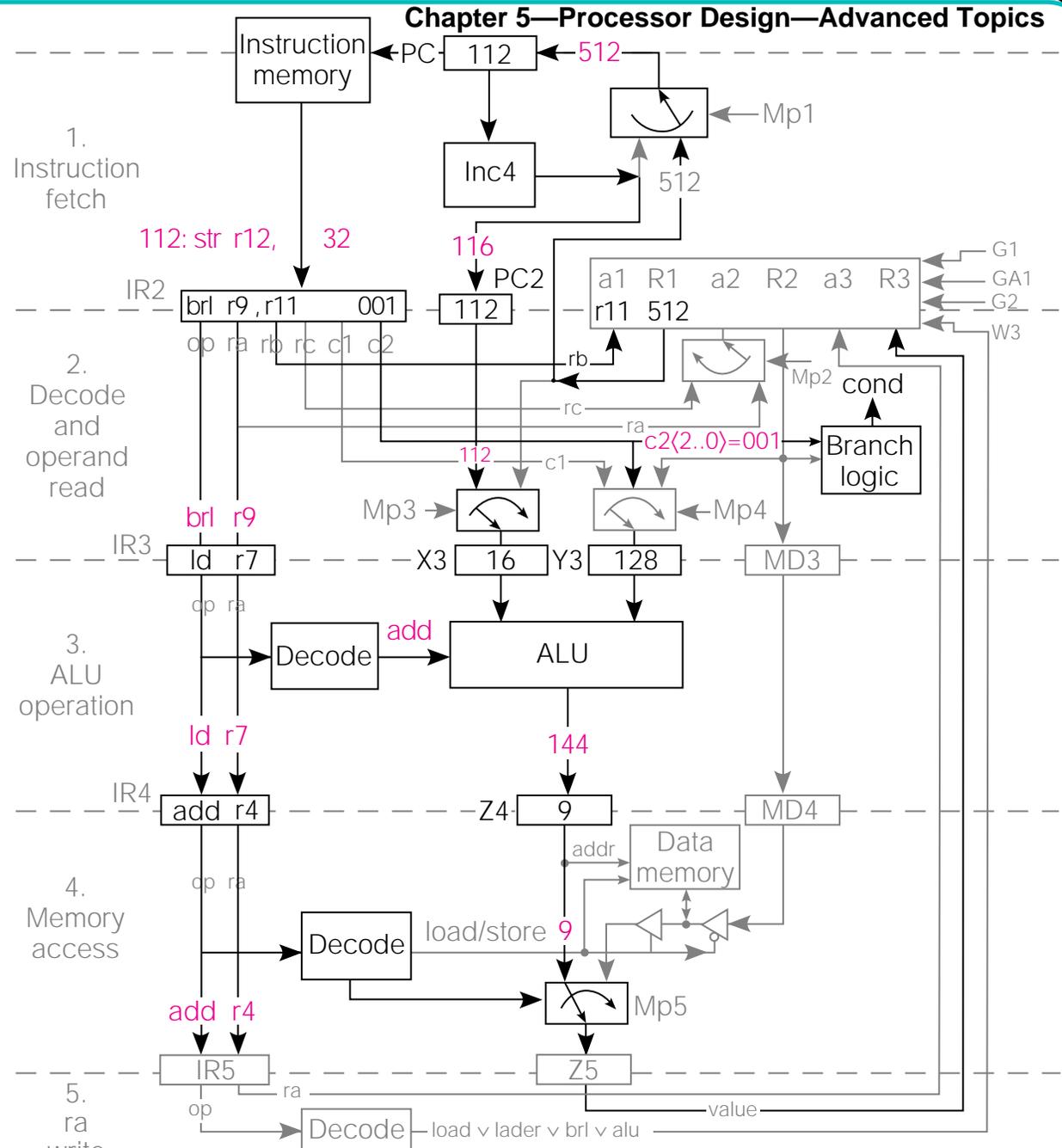


Fig 5.12 Fifth Clock Cycle: add Completes, sub Enters the Pipeline

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

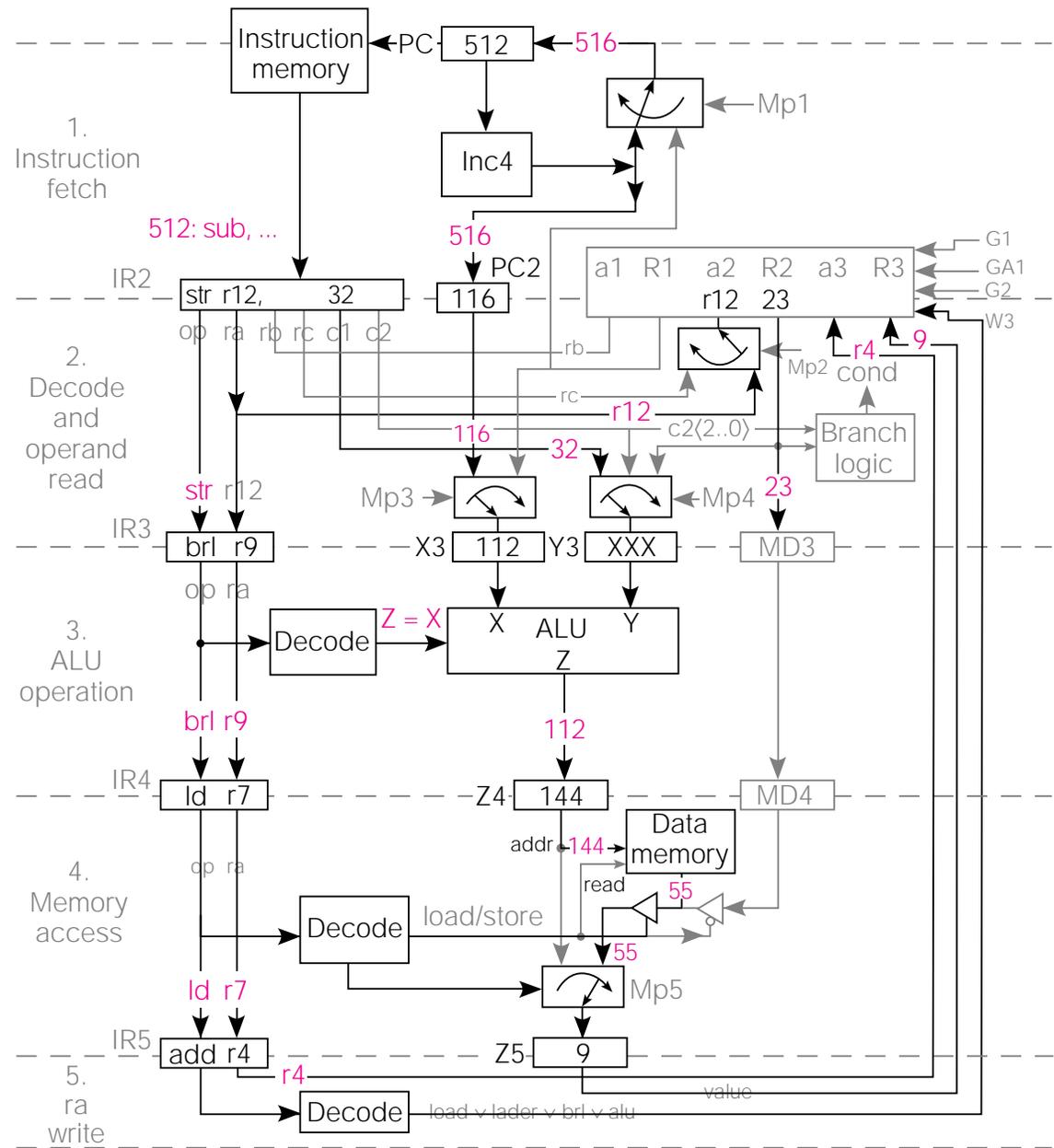
512: sub ...

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 and 2:**
 - I2 holds full instruction including any register fields and constant
 - PC2 holds the incremented PC from instruction fetch
- **Registers between stages 2 and 3:**
 - I3 holds opcode and ra (needed in stage 5)
 - X3 holds PC or a register value (for link or 1st ALU operand)
 - Y3 holds c1 or c2 or a register value as 2nd ALU operand
 - MD3 is used for a register value to be stored in memory

Functions of the Pipeline Registers in SRC (cont'd)

- **Registers between stages 3 and 4:**
 - I4 has op code and ra
 - Z4 has memory address or result register value
 - MD4 has value to be stored in data memory
- **Registers between stages 4 and 5:**
 - I5 has opcode 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 instruction 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 memory

Functions of the SRC Pipeline Stages (cont'd)

- **Stage 4: accesses data memory**
 - Passes Z4 to Z5 unchanged for nonmemory 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

Classification of Data Hazards

- A read after write hazard (RAW) arises from a flow dependence, where an instruction uses data produced by a previous one
- A write after read hazard (WAR) comes from an anti-dependence, 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 output dependence, where two parallel instructions write the same register and must do it in the order in which they were issued

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

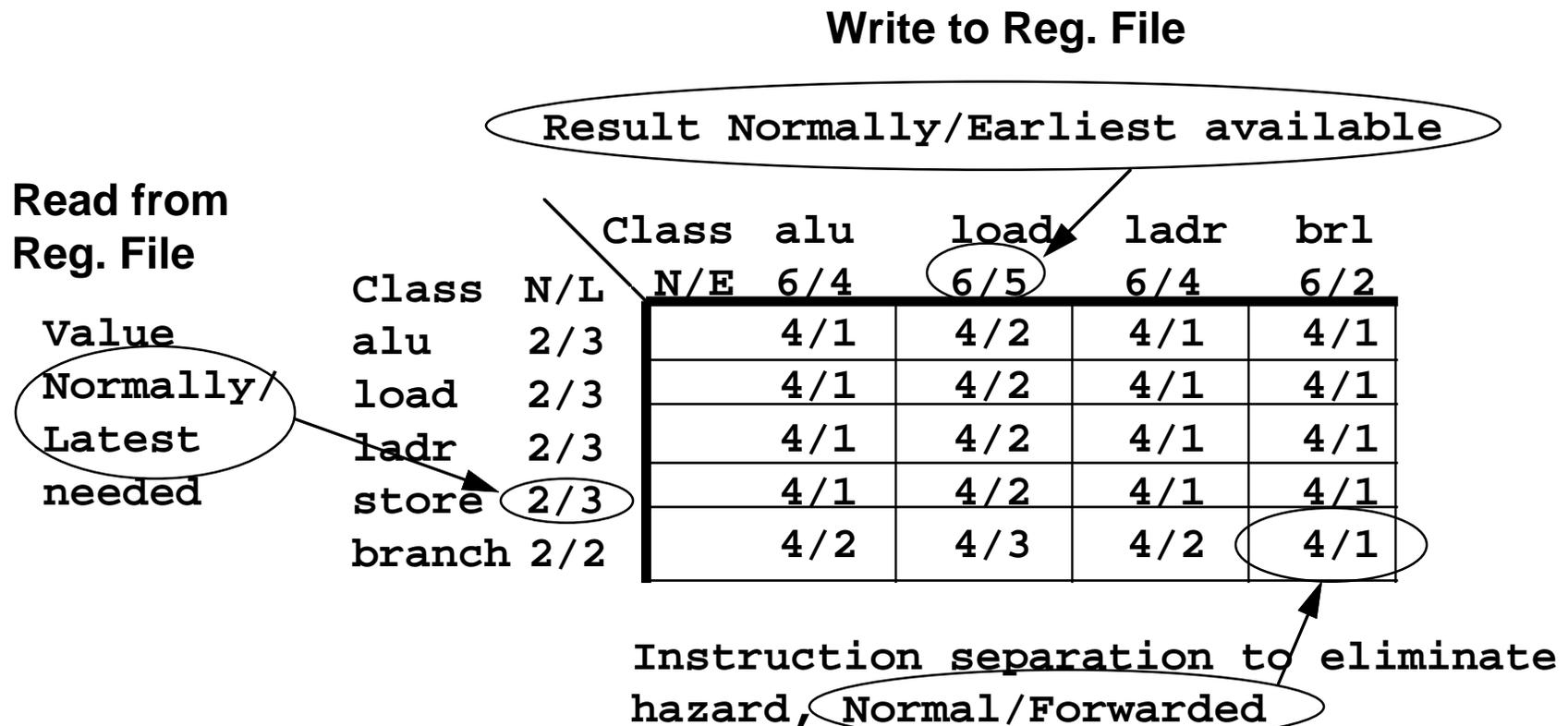
Possible Solutions to the Register Data Hazard Problem

- **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

Detecting Hazards and Dependence Distance

- To detect hazards, pairs of instructions must be considered
- Data is normally available after being written to register
- Can be made available for forwarding as early as the stage where it is produced
 - Stage 3 output for ALU results, stage 4 for memory 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

Instruction Pair Hazard Interaction



- 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 5.1

Delays Unavoidable by Forwarding

- In the Table 5.1 “Load” column, 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 memory)
 - 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 and 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 and 3 and stalls the pipeline

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

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

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

- Hazards between stages 2 & 5 require

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

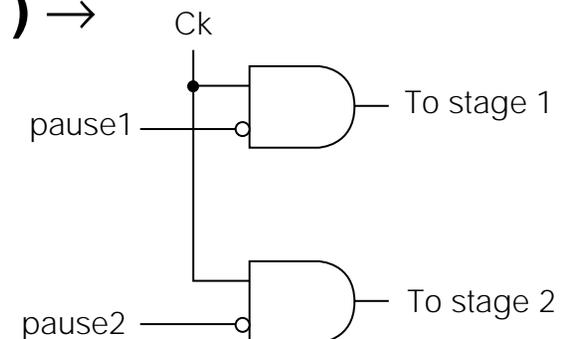


Fig 5.13 Pipeline Clocking Signals

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 instruction in stage 3 on an ALU instruction in stage 5 the equation is

$$\text{alu5} \wedge \text{alu3} \rightarrow ((\text{ra5} = \text{rb3}) \rightarrow \text{X} \leftarrow \text{Z5}:$$

$$(\text{ra5} = \text{rc3}) \wedge \neg \text{imm3} \rightarrow \text{Y} \leftarrow \text{Z5}):$$

Data Forwarding:

ALU to ALU Instruction (cont'd)

- For an ALU instruction in stage 3 depending on one in stage 4, the equation is

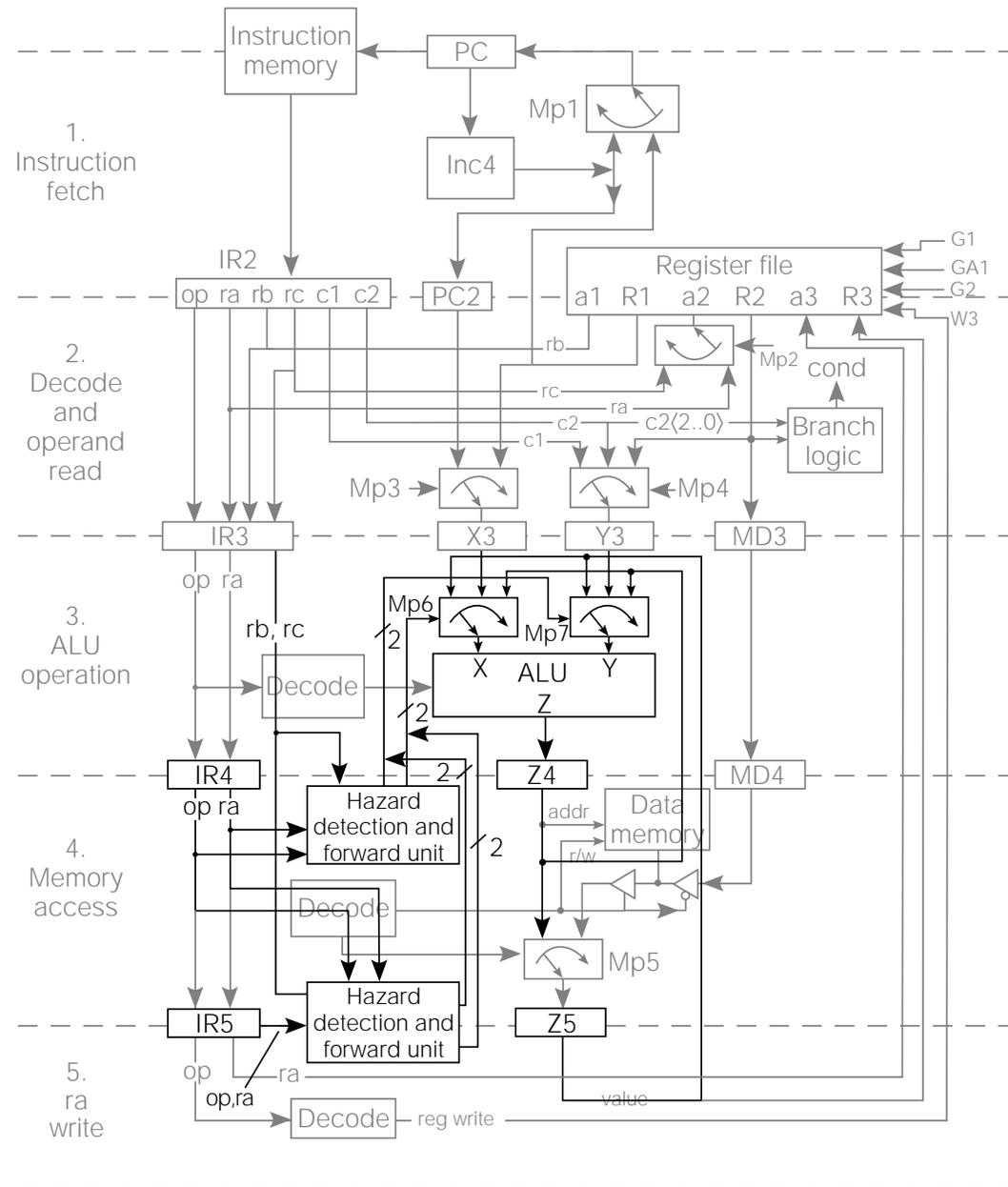
$$\text{alu4} \wedge \text{alu3} \rightarrow ((\text{ra4} = \text{rb3}) \rightarrow X \leftarrow Z4:$$

$$(\text{ra4} = \text{rc3}) \wedge \neg \text{imm3} \rightarrow Y \leftarrow Z4):$$

- 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 Hazard Detection and Forwarding

- Can be from either Z4 or Z5 to either X or Y input to ALU
- rb and 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.

```
br r4
add . . .
. . .
```

(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.

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

(3) Branch target

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

```
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**

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**

Microprogramming: Basic Idea

- Recall control sequence for 1-bus SRC

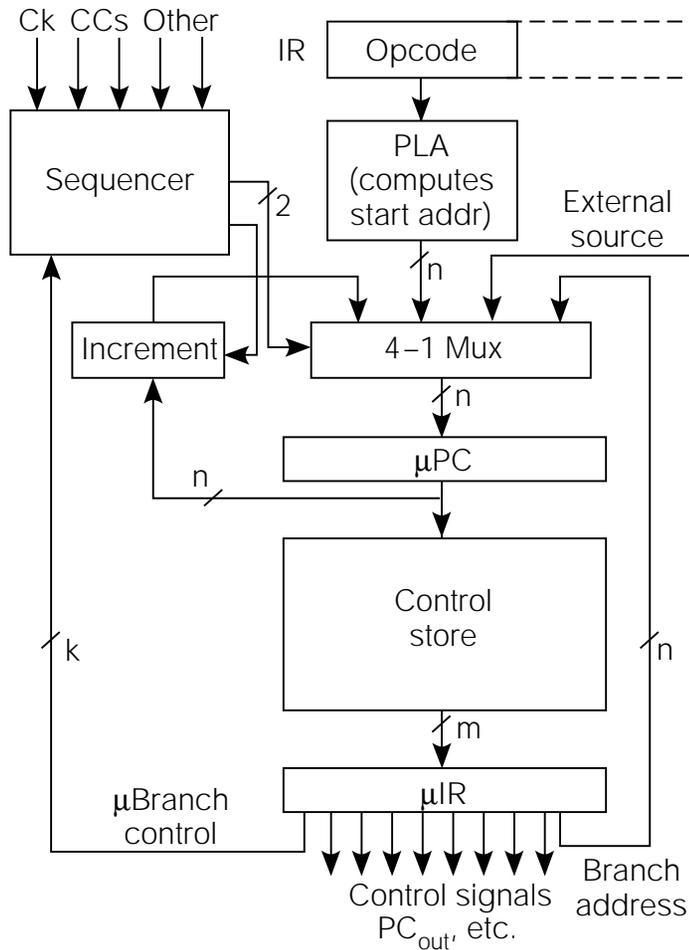
<u>Step</u>	<u>Concrete RTN</u>	<u>Control Sequence</u>
T0	$MA \leftarrow PC; C \leftarrow PC + 4;$	$PC_{out}, MA_{in}, INC4, C_{in}, Read$
T1	$MD \leftarrow M[MA]; PC \leftarrow C;$	$C_{out}, PC_{in}, Wait$
T2	$IR \leftarrow MD;$	MD_{out}, IR_{in}
T3	$A \leftarrow R[rb];$	Grb, R_{out}, A_{in}
T4	$C \leftarrow A + R[rc];$	$Grc, R_{out}, ADD, C_{in}$
T5	$R[ra] \leftarrow C;$	$C_{out}, Gra, R_{in}, End$

- 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 much 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 control store
- A control store word, or microinstruction, 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 5.16 Block Diagram of Microcoded Control Unit



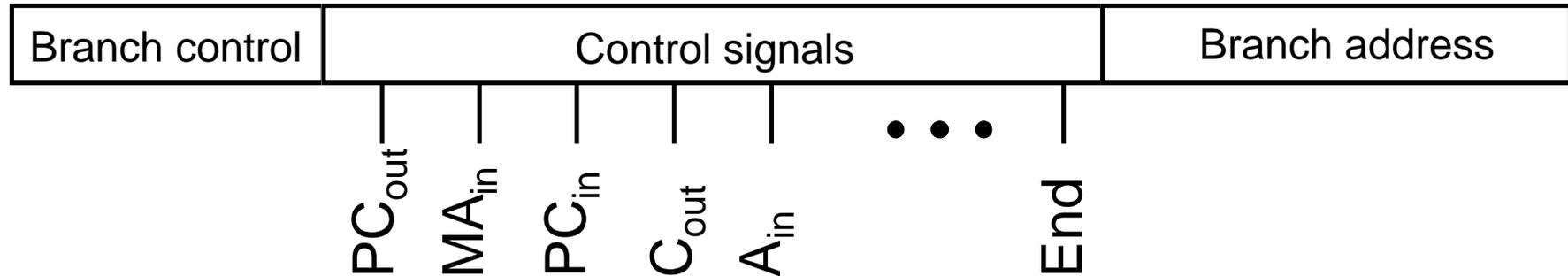
- **Microinstruction has branch control, branch address, and control signal fields**
- **Microprogram counter can be set from several sources to do the required sequencing**

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— $\mu\text{PC} + 1$
 - PLA output—start address for a macroinstruction
 - Branch address from $\mu\text{instruction}$
 - External source—say for exception or reset
- Micro conditional branches can depend on condition codes, data path state, external signals, etc.

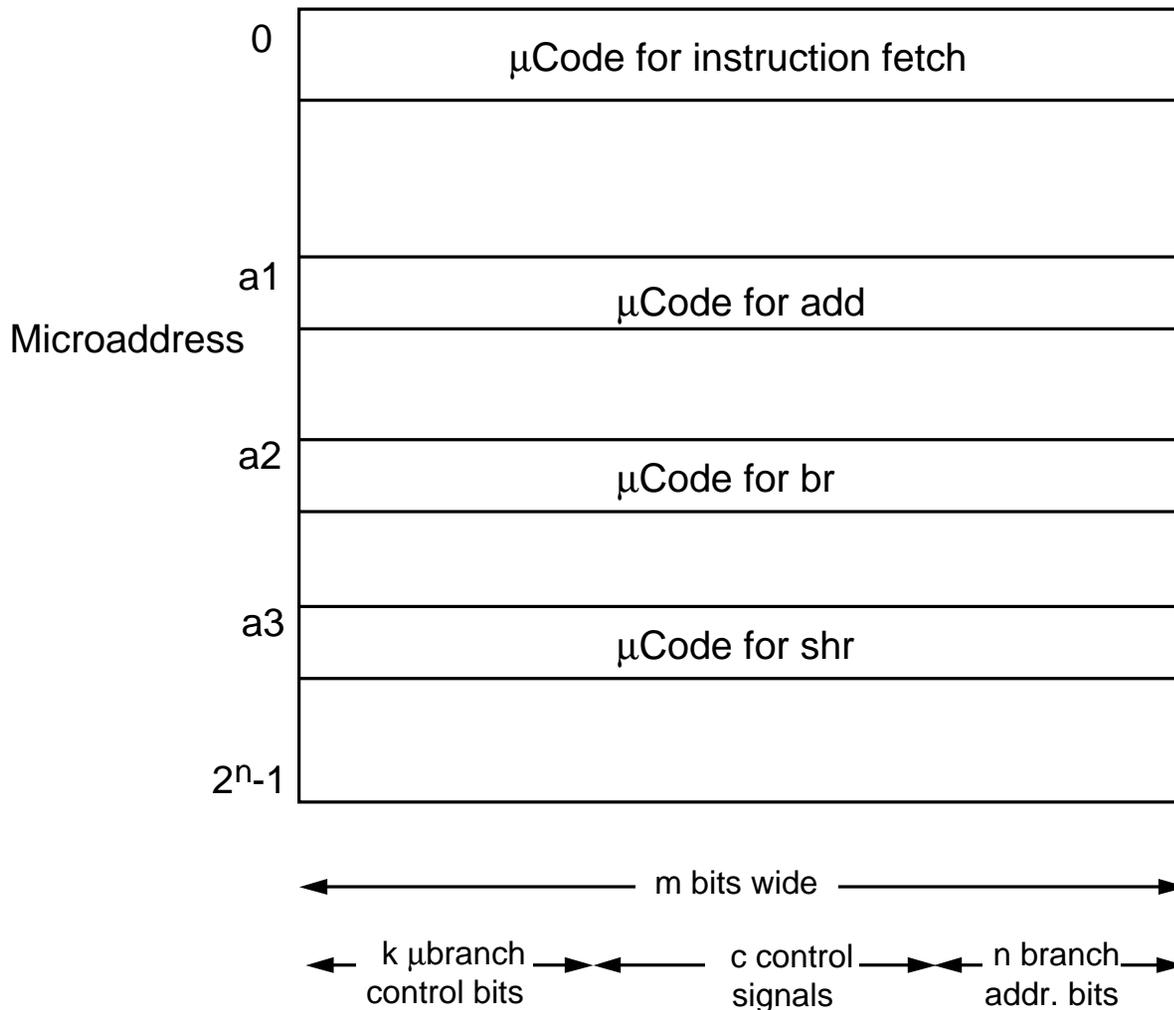
Contents of a Microinstruction

Microinstruction format



- 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 $\mu PC + 1$

Fig 5.17 The Control Store



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

Tbl 5.2 Control Signals for the add Instruction

Address	Branch Control	PC _{out}	C _{out}	MD _{out}	R _{out}	MA _{in}	C _{in}	PC _{in}	IR _{in}	A _{in}	R _{in}	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	0	1	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	0	1	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 μ control from 103 to 200 uses a kind of μ branch

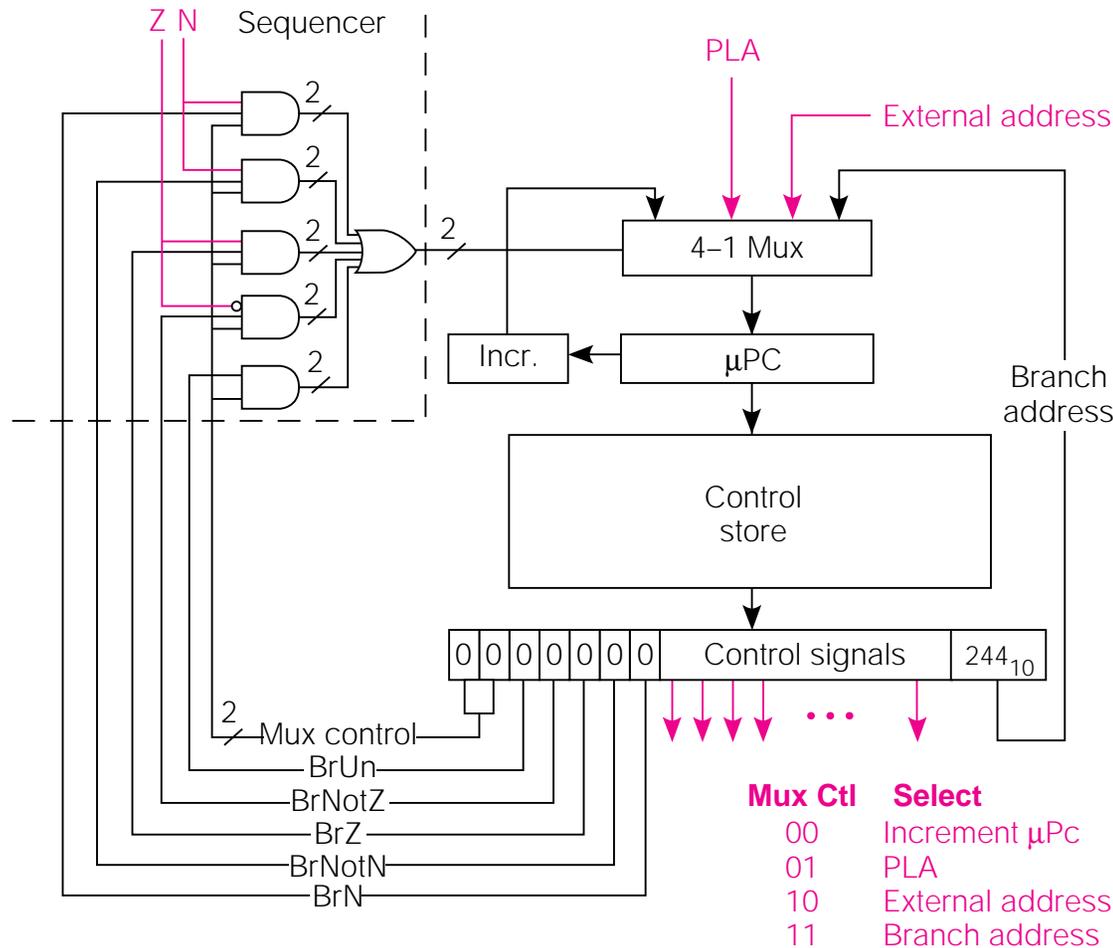
Uses for μ branching in the Microprogrammed Control Unit

- (1) Branch to start of μ code for a specific inst.
 - (2) Conditional control signals, e.g. $CON \rightarrow PC_{in}$
 - (3) Looping on conditions, e.g. $n \neq 0 \rightarrow \dots Goto6$
- Conditions will control μ branches instead of being ANDed with control signals
 - Microbranches are frequent and control store addresses are short, so it is reasonable to have a μ branch address field in every μ instruction

Illustration of μ branching Control Logic

- We illustrate a μ branching control scheme by a machine having condition code bits N and Z
- Branch control has 2 parts:
 - (1) selecting the input applied to the μ PC and
 - (2) specifying whether this input or μ PC + 1 is used
- We allow 4 possible inputs to μ PC
 - The incremented value μ PC + 1
 - The PLA lookup table for the start of a macroinstruction
 - An externally supplied address
 - The branch address field in the μ instruction word

Fig 5.18 Branching Controls in the Microcoded Control Unit



- **5 branch conditions**
 - NotN
 - N
 - NotZ
 - Z
 - Unconditional
- **To 1 of 4 places**
 - Next μ instruction
 - PLA
 - External address
 - Branch address

Some Possible μ branches Using the Illustrated Logic (Refer to Tbl 5.3)

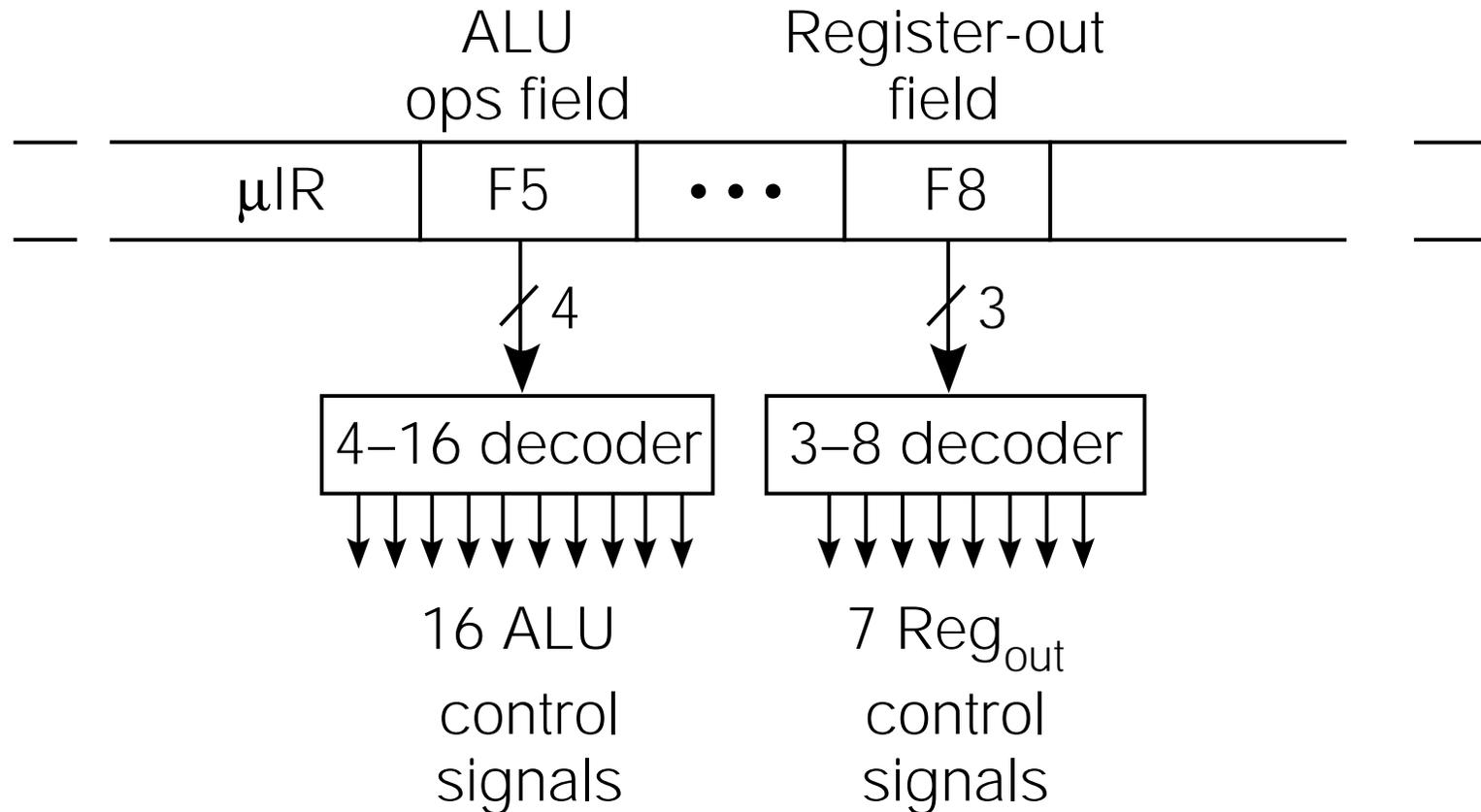
Mux Ctl	BrUn	BrNotZ	BrZ	BrNotN	BrN	Control Signals	Branch Address	Branching action
00	0	0	0	0	0	•••	XXX	None—next instruction
01	1	0	0	0	0	•••	XXX	Branch to output of PLA
10	0	0	1	0	0	•••	XXX	Br if Z to Extern. Addr.
11	0	0	0	0	1	•••	300	Br if N to 300 (else next)
11	0	0	0	1	0	0•••0	206	Br if \bar{N}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

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

Horizontal versus Vertical Microcode Schemes

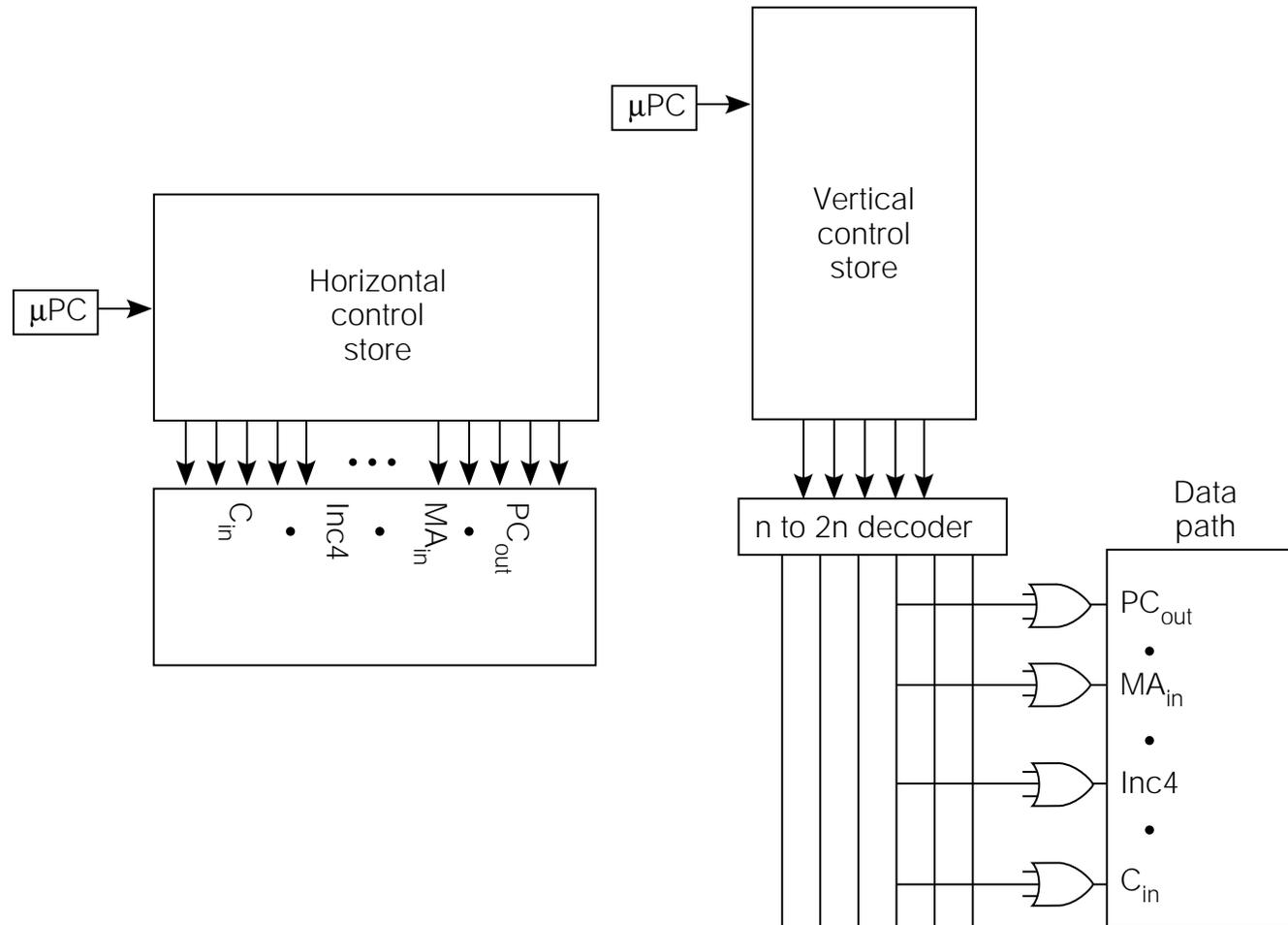
- In horizontal microcode, each control signal is represented by a bit in the μ instruction
- In vertical microcode, a set of true control signals is represented by a shorter code
- The name horizontal implies fewer control store words of more bits per word
- Vertical μ code only allows RTs in a step for which there is a vertical μ instruction code
- Thus vertical μ code may take more control store words of fewer bits

Fig 5.19 A Somewhat Vertical Encoding



- **Scheme would save $(16 + 7) - (4 + 3) = 16$ bits/word in the case illustrated**

Fig 5.20 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 μ branches BrCON, Brn = 0, and Brn $\neq 0$
- We adopt the clocking logic of Fig. 4.14
- 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

Tbl 5.4 The add Instruction

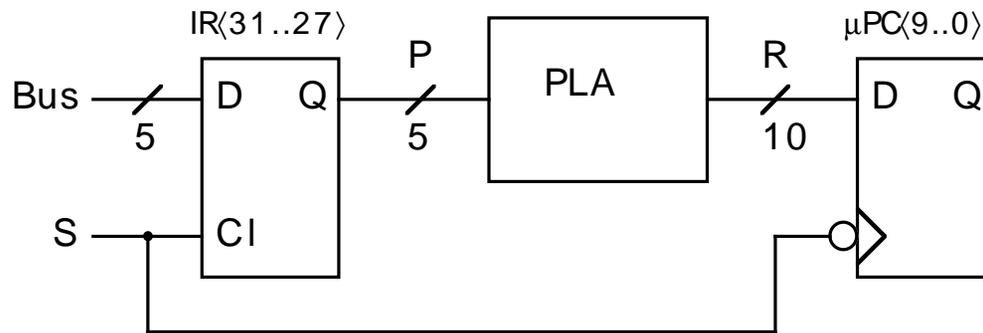
Addr.	Mux Ctl	BrUn	BrCON	Brn≠0	Brn=0	End	PCout	MA _{in}	Other Control Signals	Br Addr.	Actions
100	00	0	0	0	0	0	1	1	...	XXX	$MA \leftarrow PC; C \leftarrow PC+4;$
101	00	0	0	0	0	0	0	0	...	XXX	$MD \leftarrow M[MA]; PC \leftarrow C;$
102	01	1	0	0	0	0	0	0	...	XXX	$IR \leftarrow MD; \mu PC \leftarrow PLA;$
200	00	0	0	0	0	0	0	0	...	XXX	$A \leftarrow R[rb];$
201	00	0	0	0	0	0	0	0	...	XXX	$C \leftarrow A + R[rc];$
202	11	1	0	0	0	1	0	0	...	100	$R[ra] \leftarrow C; \mu PC \leftarrow 100;$

- **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

- For the input to the PLA to be correct for the μ branch in 102, it has to come from MD, not IR
- An alternative is to use see-through latches for IR so the opcode can pass through IR to PLA before the end of the clock cycle

See-Through Latch Hardware for IR So μ PC Can Load Immediately



- Data must have time to get from MD across Bus, through IR, through the PLA, and satisfy μ PC set up time before trailing edge of S

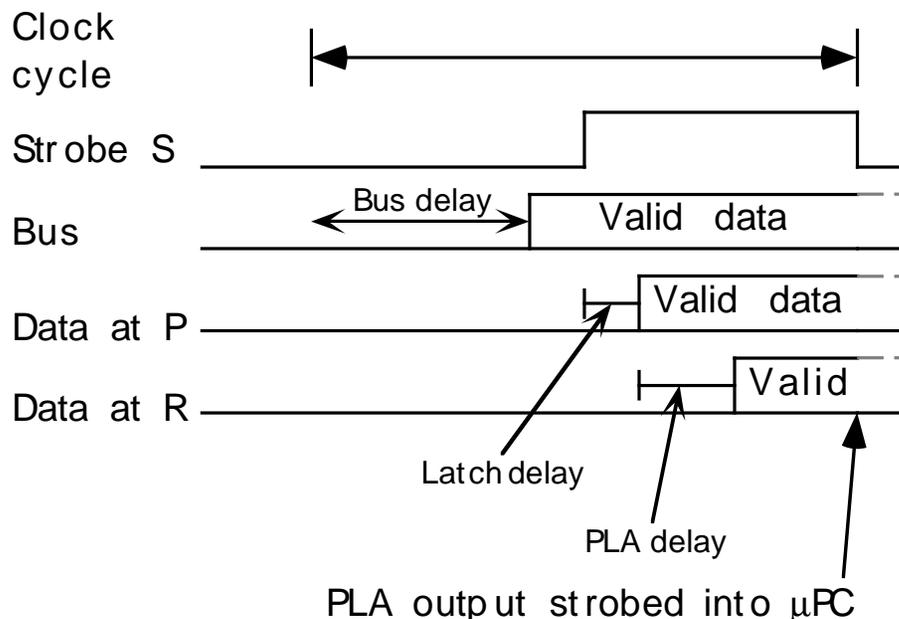
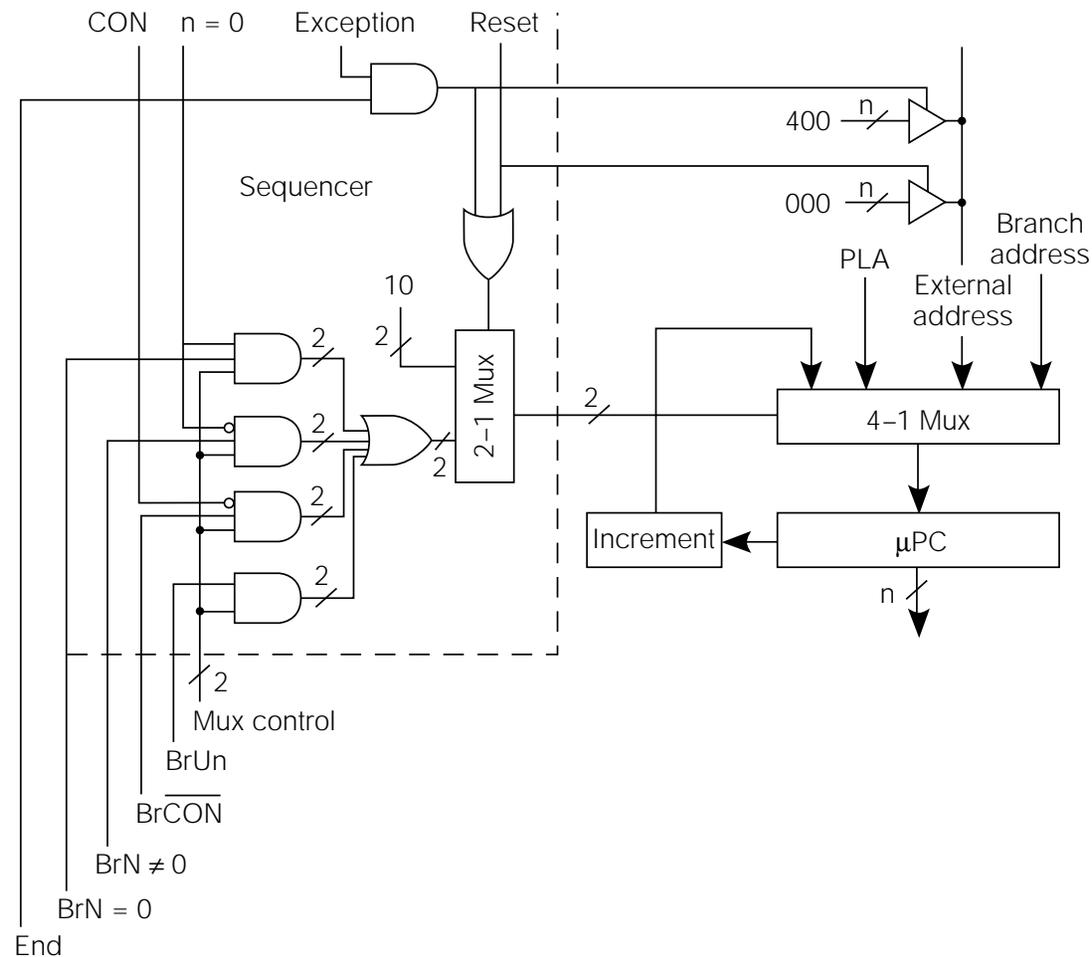


Fig 5.21 SRC Microcode Sequencer



Tbl 5.6 Somewhat Vertical Encoding of the SRC Microinstruction

F1	F2	F3	F4	F5	F6	F7	F8	F9
Mux Ct I	Branch control	End	Out signals	In signals	Misc.	Gate regs.	ALU	Branch address
00 01 10 11	000 BrUn 001 Br-CON 010 BrCON 011 Br 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
2 bits	3 bits	1 bit	3 bits	3 bits	3 bits	2 bits	4 bits	10 bits

Other Microprogramming Issues

- **Multiway branches: often an instruction can have 4–8 cases, say address modes**
 - **Could take 2–3 successive μ branches, i.e. clock pulses**
 - **The bits selecting the case can be ORed into the branch address of the μ instruction to get a several way branch**
 - **Say if 2 bits were ORed into the 3rd and 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 multiway branch in one clock**
- **A hardware push-down stack for the μ PC can turn repeated μ sequences into μ subroutines**
- **Vertical μ code can be implemented using a horizontal μ engine, 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 instruction per clock**
- **Forwarding, branch delay slot, and load delay slot are steps in approaching this goal**
- **More than one issue per clock is possible, but beyond the scope of this text**
- **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**