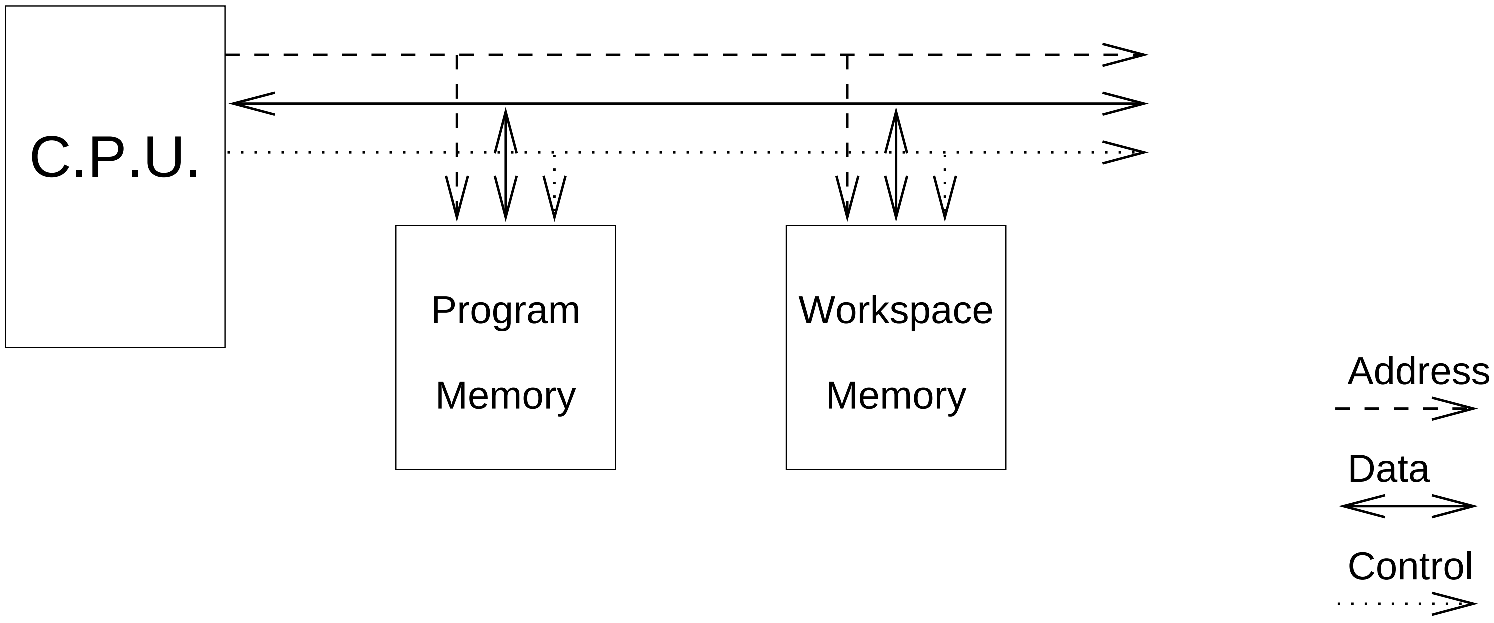
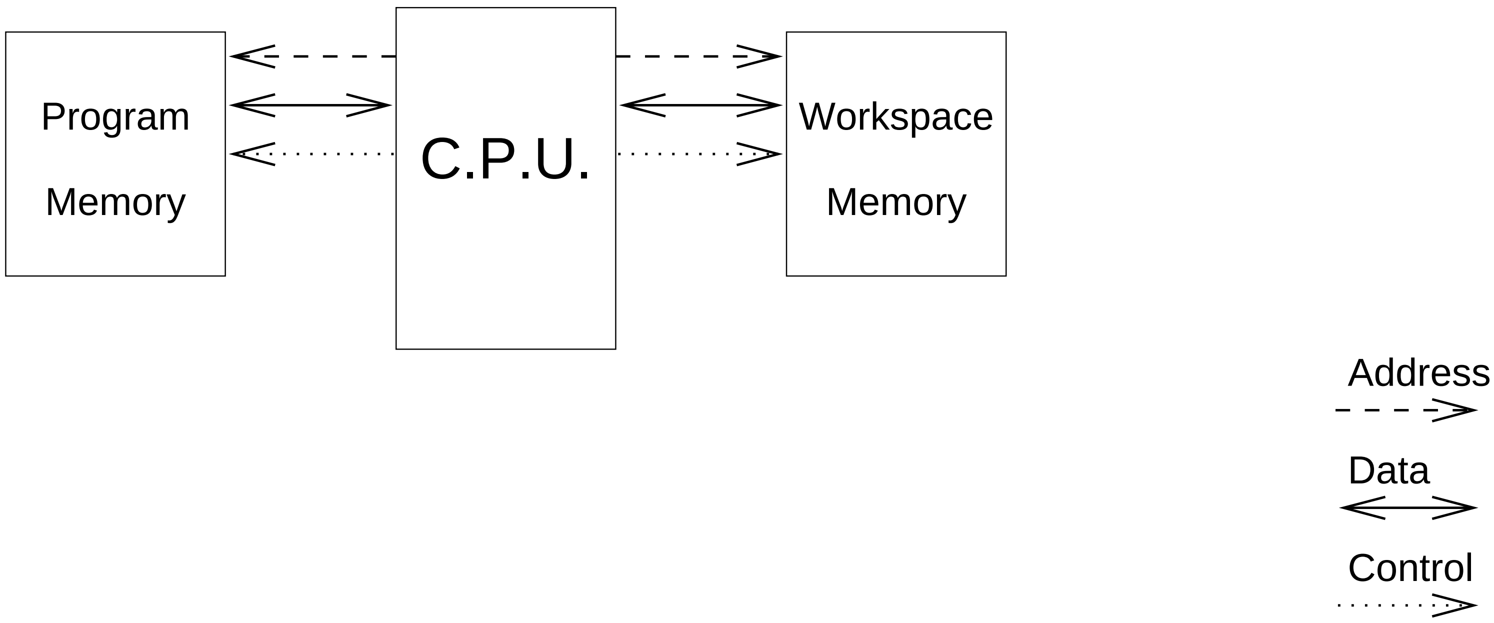
Princeton Architecture for Serial Computers



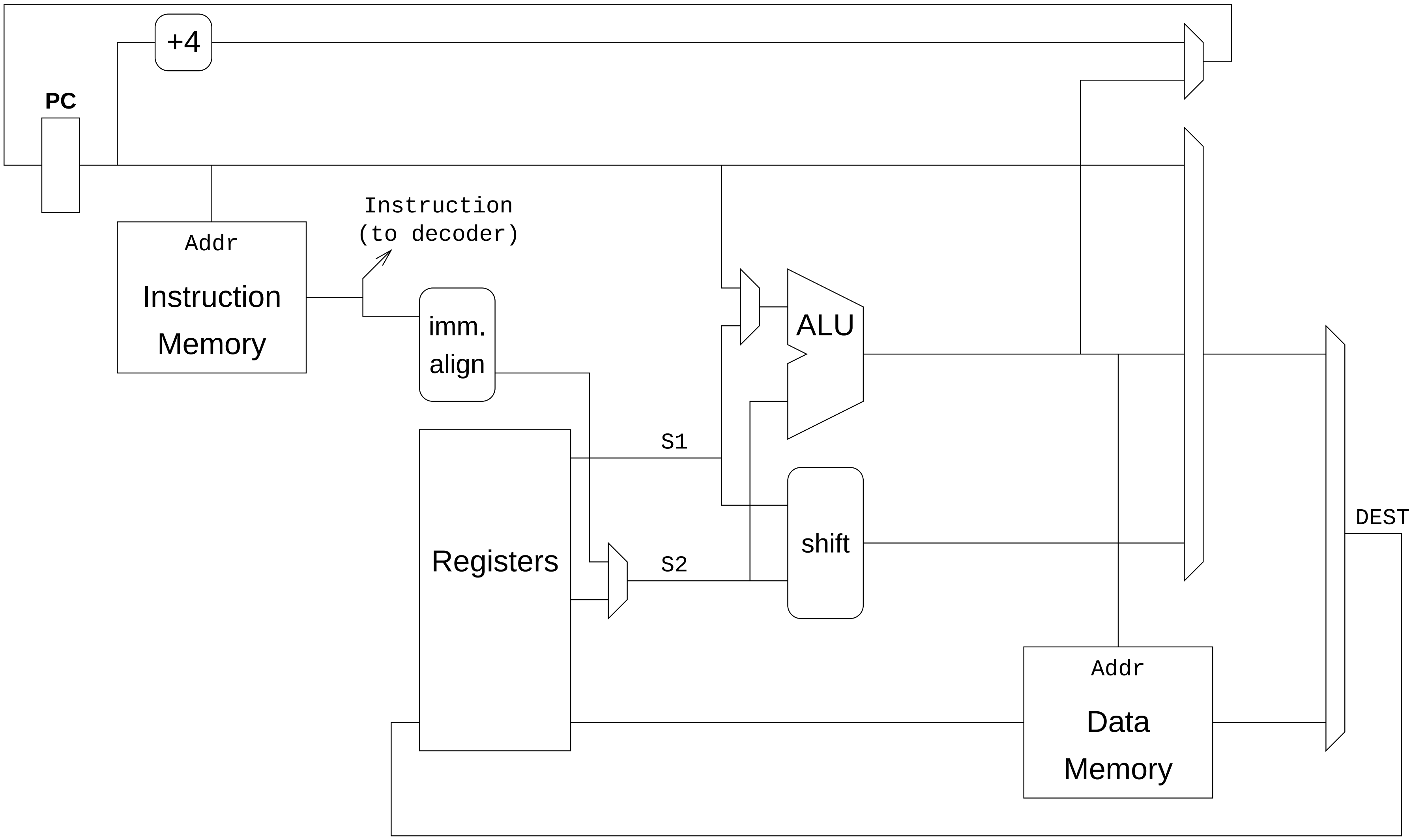
- The Princeton Architecture uses a common set of buses for all components.

Harvard Architecture for Serial Computers



- The Harvard Architecture uses different sets of buses for program and workspace memories.

1 c.p.i. Harvard Architecture for SPARC



Instruction Cycle

To support a throughput of 1 instruction every cycle we must ensure that no bus or functional unit is used more than once in the instruction cycle.

- Separate incrementer for PC

  although instruction set allows for the ALU to be used for data address calculation and branch address calculation

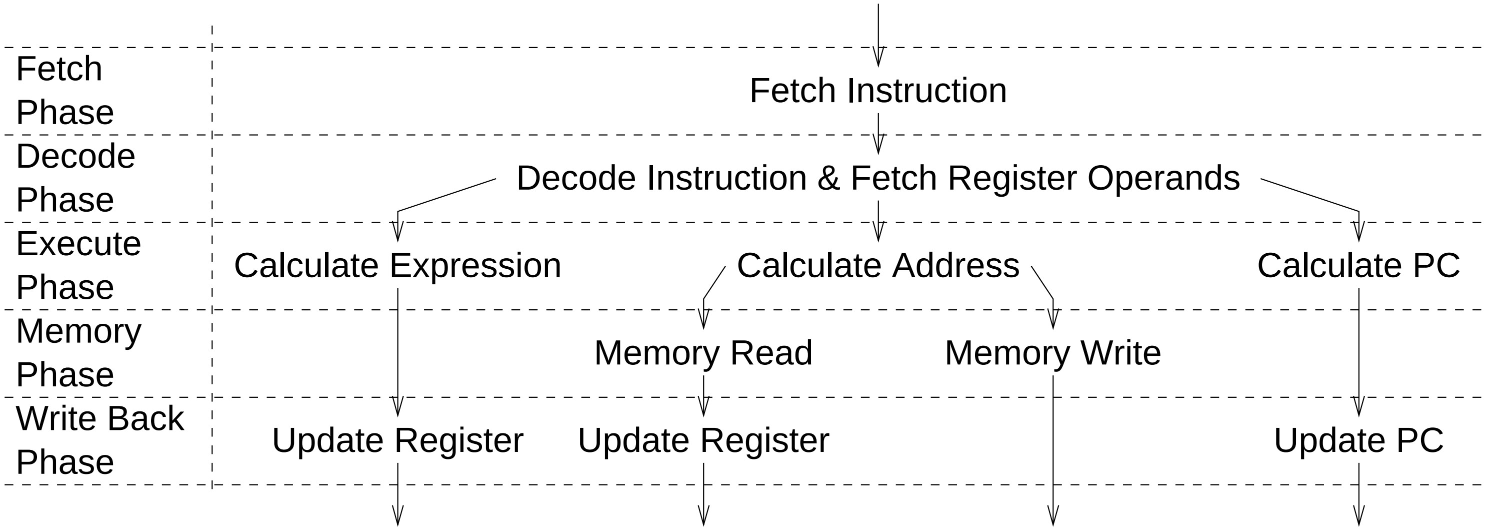
- 4 port register file

  allows three reads and one write in any one cycle

To illustrate the allocation of resources, we shall divide our instruction cycle into five phases:

- IF (Instruction Fetch)
- ID (Instruction Decode/Register Fetch)
- EXE (Execute)
- MEM (Memory)
- WB (Write Back)

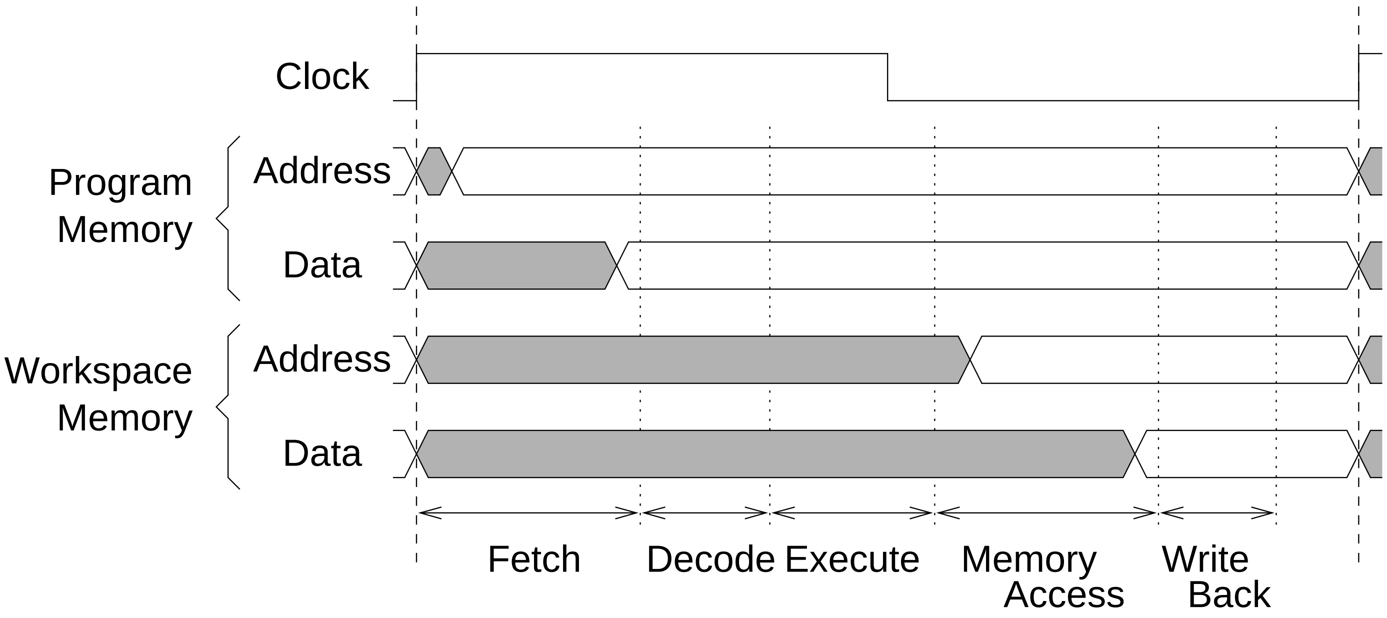
Instruction Cycle



No unit is used more than once for each instruction[1], the whole instruction may be completed in a *single clock cycle*.

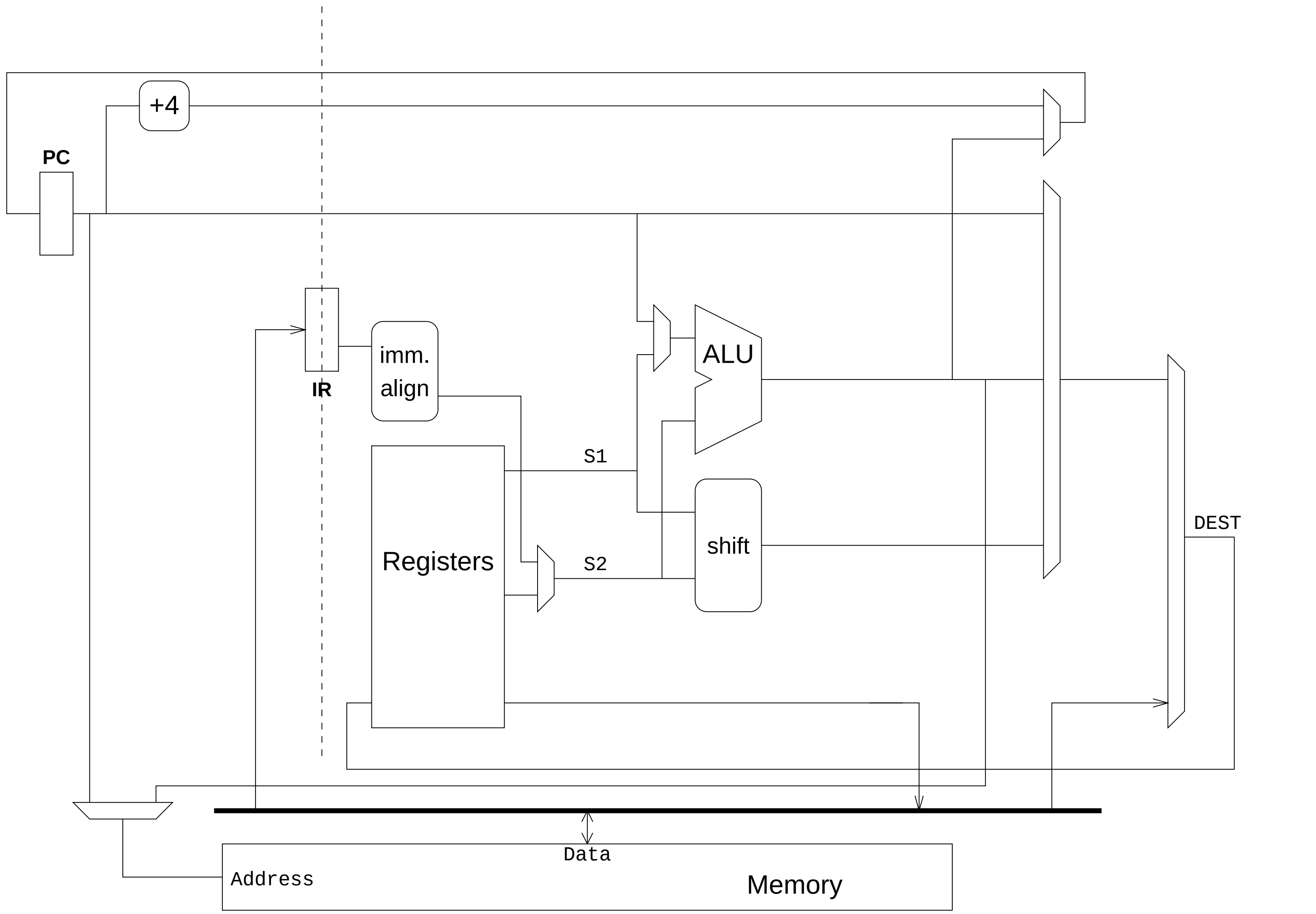
[1]1 program memory access, 1 instruction decode, 1 ALU calculation, 1 (or 0) data register update, 1 (or 0) workspace memory access

Instruction Cycle

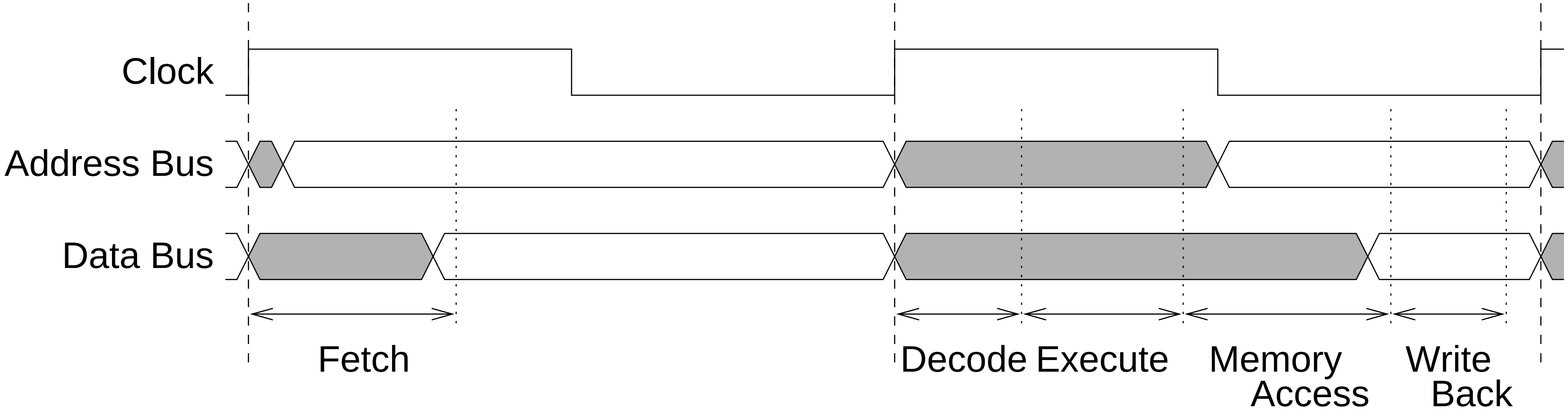


- The period of the clock must be long enough to accommodate the worst case timings for the worst case instruction.

2 c.p.i. Princeton Architecture for SPARC

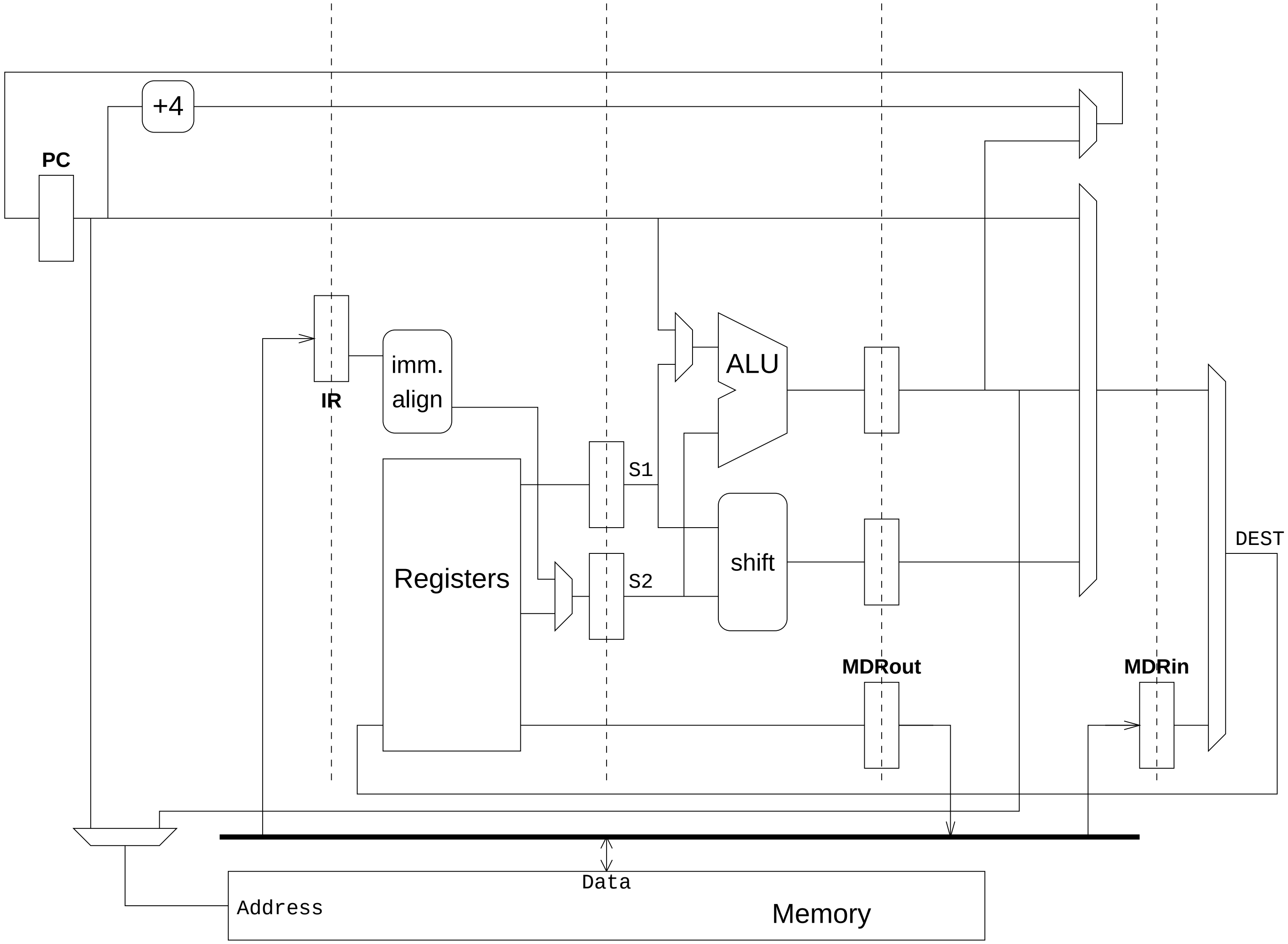


Instruction Cycle

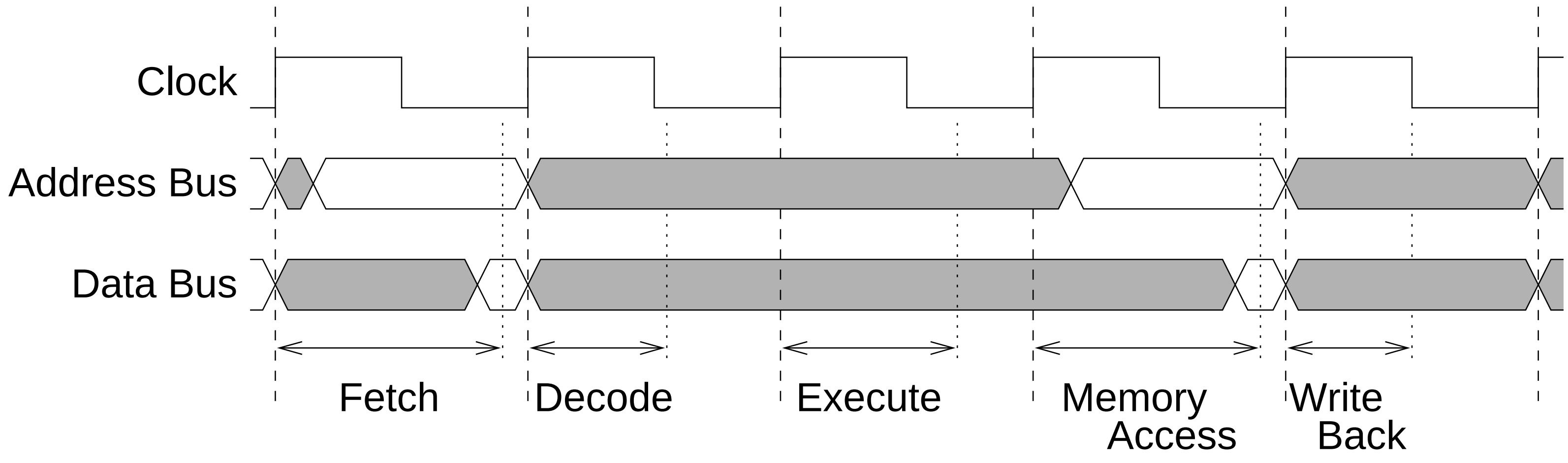


- Memory is shared using Time-Slice Multiplexing
- 2 **C**ycles **P**er **I**nstruction
  - Cycle 1
    The controller fetches an instruction from the common memory.
    The instruction is latched in the new *Instruction Register*
  - Cycle 2
    The datapath may access data in the common memory.

5 c.p.i. Princeton Architecture for SPARC



Instruction Cycle



- 5 **C**ycles **P**er **I**nstruction
  - Additional registers are required to store intermediate results
    S1 and S2 registers are updated at the end of cycle 2.
    ALU output, Shifter output and MDR output registers are updated at the end of cycle 3.
    MDR input register is updated at the end of cycle 4.
  - A better use of available time where memory access doesn't dominate
  - Latching overheads are more significant for shorter clock cycles

Processor Performance

- Let us consider the performance of our simple machines:

$$Work\ Rate = \frac{Instruction\ Rate}{Instructions\ per\ Task}$$

  – The *Instructions per Task* value will be determined by the power of our instruction set and how well it matches the task to be performed.

$$Instruction\ Rate\ (Mips) = \frac{Clock\ Frequency\ (MHz)}{Cycles\ per\ Instruction}$$

  – The *Cycles per Instruction* value is an average over the whole task. For the moment we shall consider this to be a constant[2] for the machines we have described, although optimizations are possible for the 5 c.p.i. architecture.

[2] 1, 2 or 5 cycles depending on the implementation

Abuse of Statistics

- Mips
  - The *Instruction Rate* of a machine is frequently given as a measure of its performance. Since an average *Instruction Rate* may depend on the set of instructions chosen the maximum *Instruction Rate* is quoted.

    Here we can take *Mips* to mean *Meaningless Indication of Processor Speed*, the best we can hope for is an indication of the relative performance of machines sharing the same instruction set and a similar architecture.

- VAX Mips
  - Relative Mips value w.r.t. to VAX 11/780.

    This is better but will still dependent on the task chosen for the comparison.

- Benchmarking
  - Competition between those who write benchmark programs and those who modify compilers and architectures in order to achieve favourable results.

Sequential Nature of the Sequential Computer

Taking a simplified view of instruction execution:

- Fetch
- Decode
- Execute
- Memory Access
- Write Back

Repeat once for each instruction.

Although the process appears inherently sequential[3], there is room for some tasks to be done in parallel.

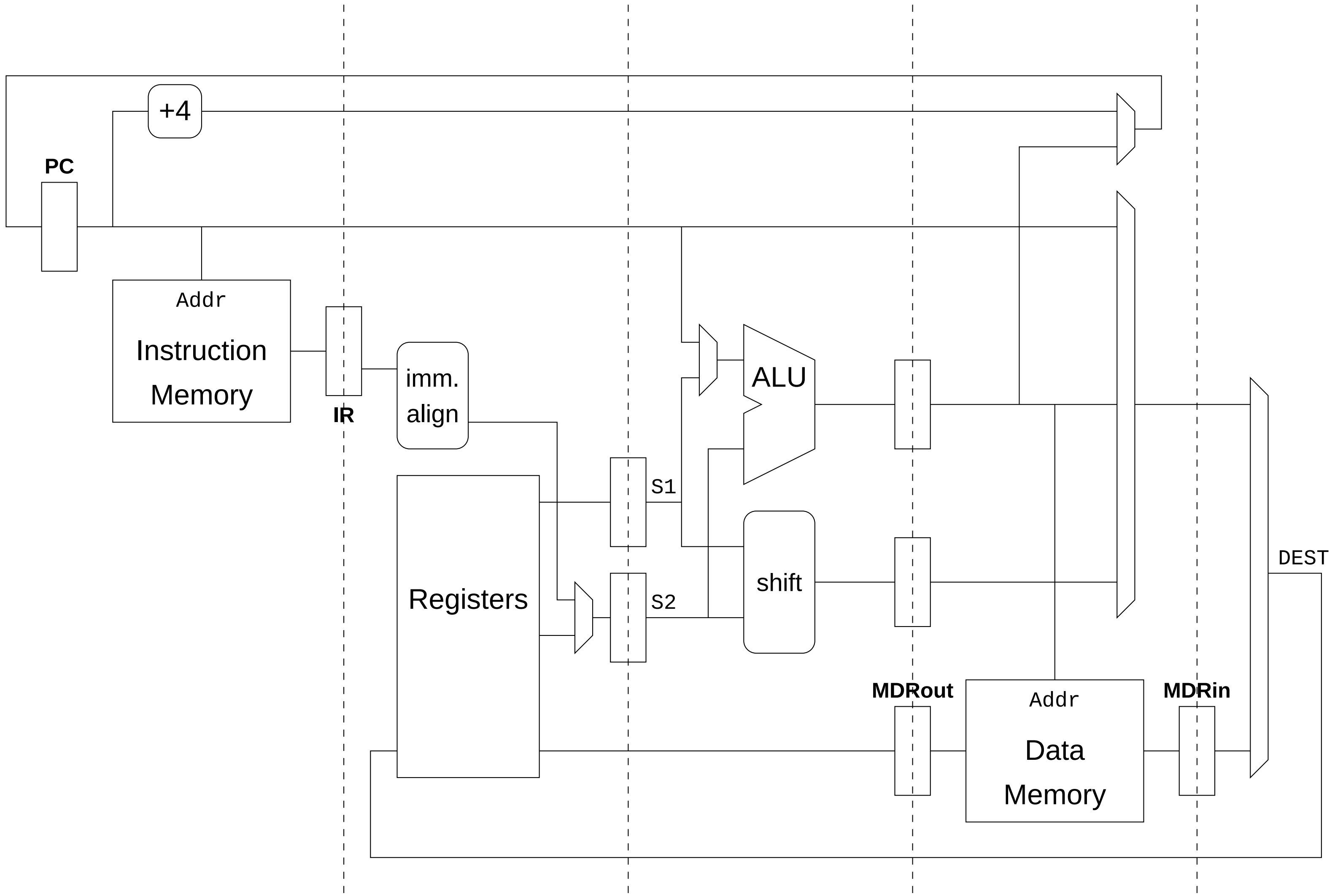
[3] each operation depends on the completion of the previous operation

Pipelining

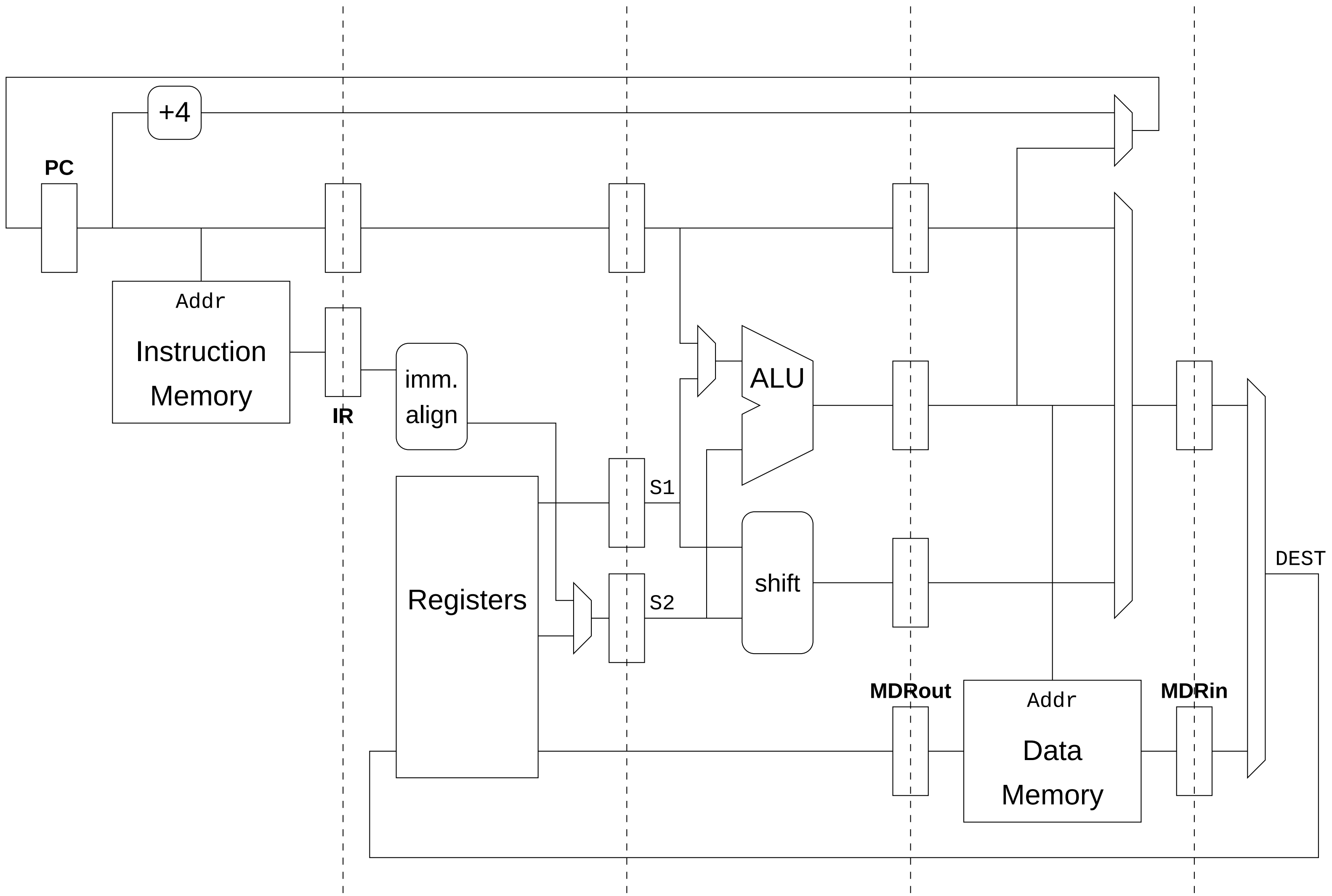
- We begin one operation before the previous operation has completed.
  - Although each individual instruction will take the same length of time to complete, a group of instructions may be completed in a much shorter time due to overlap.
  - The increase in performance arises since we are making more efficient use of our hardware. More of the hardware is kept busy for more of the time.
  - We have a limited form of *concurrency*

- A 5 c.p.i. Harvard Architecture can be modified to produce a 5 stage pipeline machine:

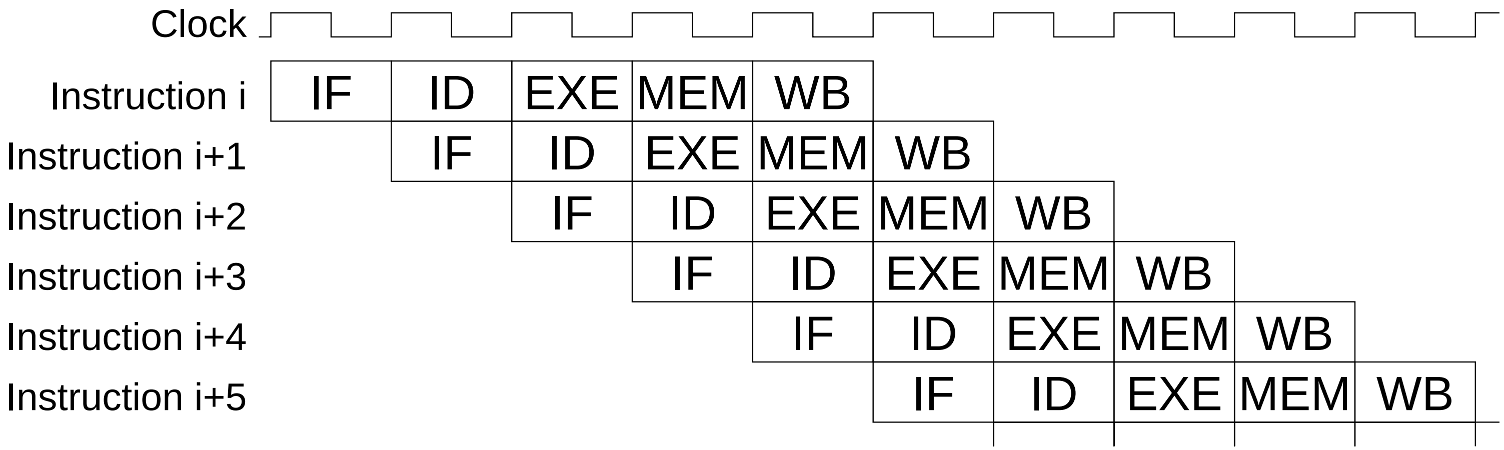
5 c.p.i. Harvard Architecture



5 Stage Pipeline Harvard Architecture

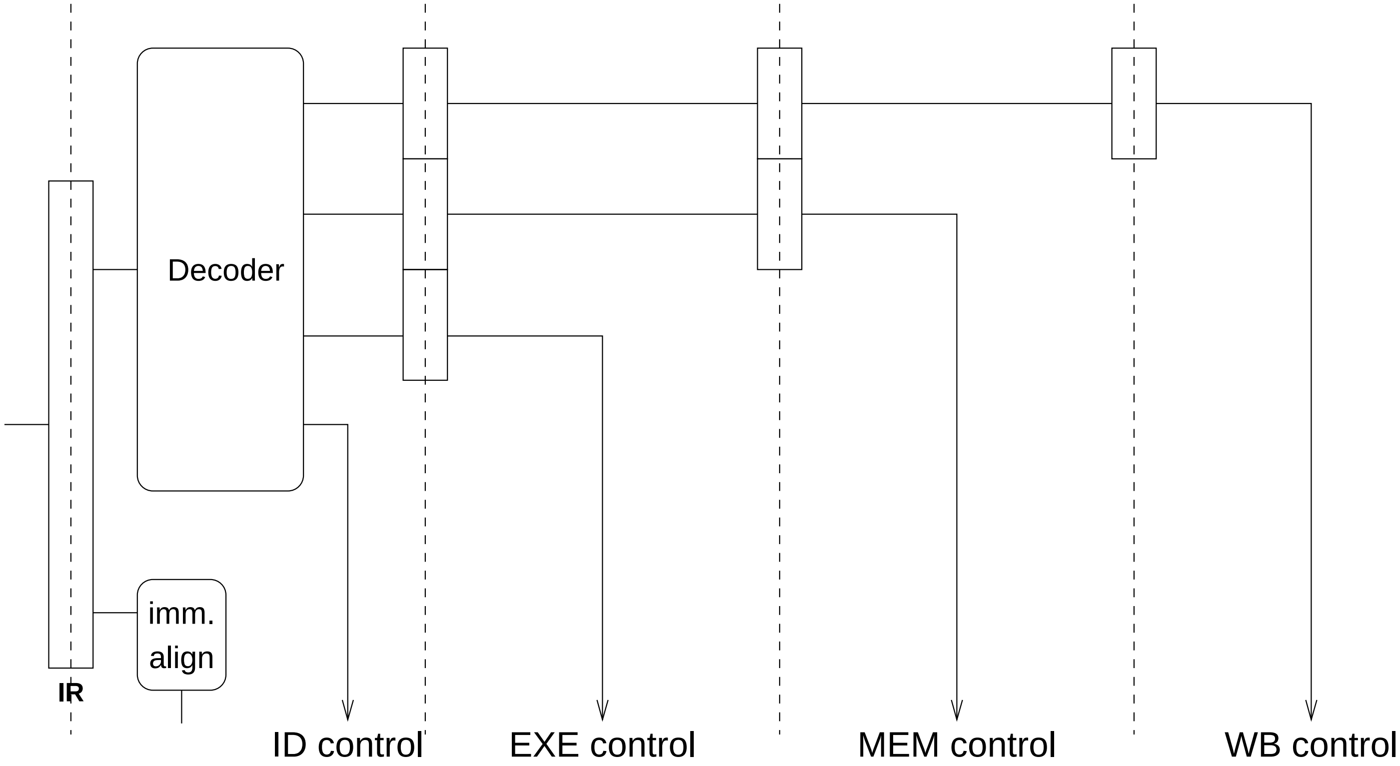


5 Stage Pipeline Operation



- A new instruction is fetched on every clock cycle.
- The machine completes five part instructions every clock cycle giving an average of one c.p.i.

A simple control unit provides control signals to all units



Note that the registers block must receive control signals from ID, EXE and WB since its four ports are connected to different pipeline stages.

Pipeline Hazards

The requirement to start a new instruction every clock cycle cannot be met where the modified order of execution will affect the final outcome. Such a situation is described as a *pipeline hazard*

- *Control Hazards* arise from the pipelining of control transfer instructions which modify the Program Counter.

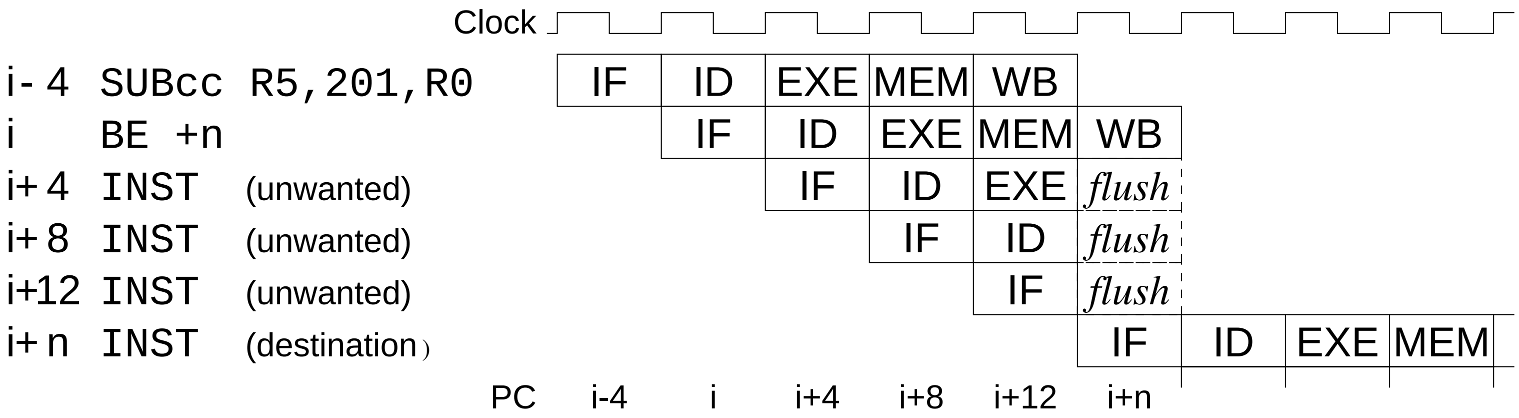
- *Structural Hazards* arise from resource conflicts where the hardware cannot support the requirements of two overlapping instructions.

- *Data Hazards* arise when an instruction depends on the result of a previous instruction which has not yet completed due to the overlapping of instructions.

Control Hazards

- Let us consider the effect of a branch on our pipeline SPARC.

  Since the PC is modified at the end of the MEM stage we fetch three additional unwanted instructions before the branch destination is known.



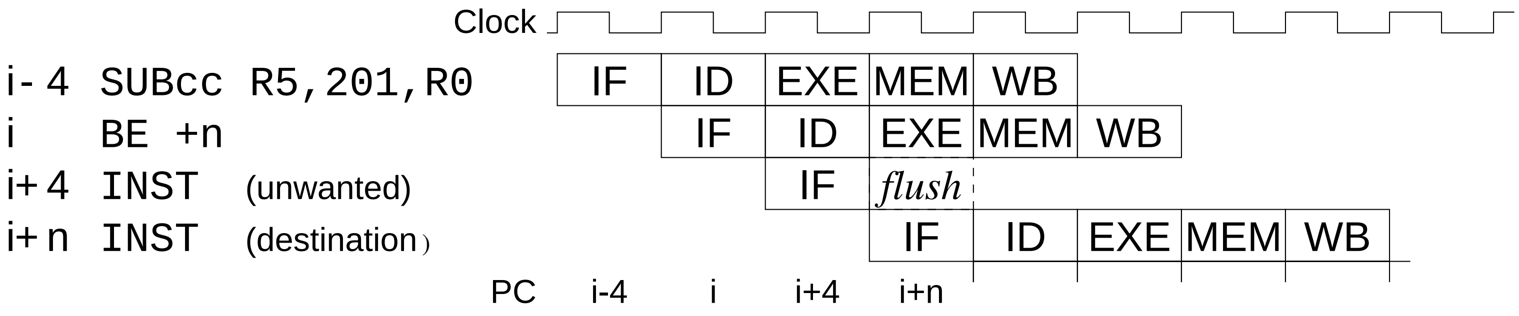
  - must *flush* pipeline of unwanted instructions
  - these instructions have no effect since they are caught before the MEM and WB stages.
  - where a conditional branch is not taken, no *flush* occurs and execution continues as before.

Control Hazards

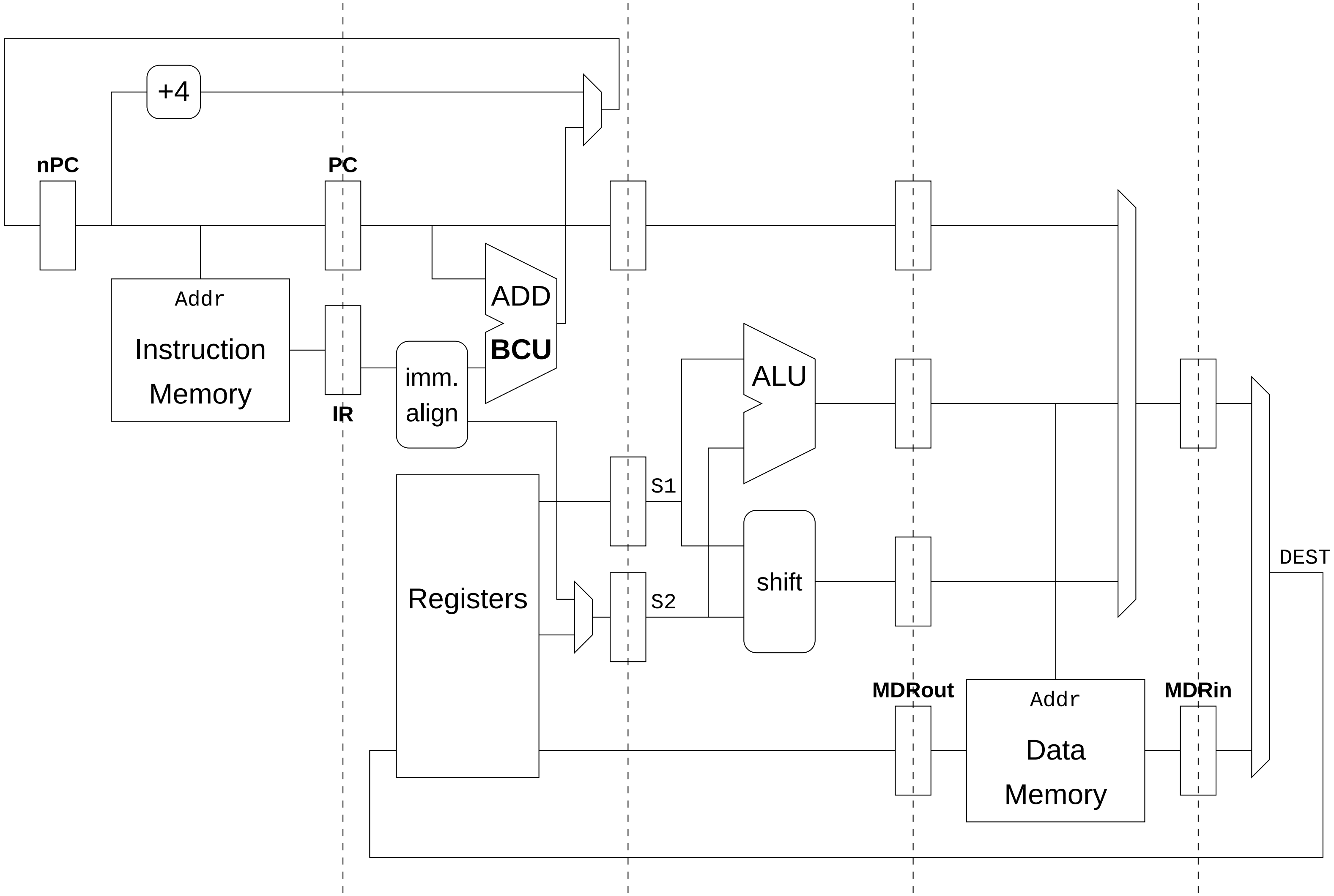
- Reducing the branch penalty

  One of the simplest ways to reduce the branch penalty is to calculate the destination address earlier.

  With the addition of a dedicated Branch Calculation Unit adder for PC relative control transfers (`CALL` and `Bicc`), we can calculate the branch address in the ID stage.



SPARC



+4
nPC
PC
Addr
Instruction
Memory
IR
ADD
BCU
imm.
align
ALU
S1
S2
shift
Registers
DEST
MDRout
Addr
Data
Memory
MDRin

Control Hazards

This technique of flushing instructions when a branch is taken is actually a simple form of branch prediction:

- Static branch prediction - predict branch not taken

Two other simple schemes exist:

- Always stall

  With this technique we stall instruction fetch until we know the branch destination. For a conditional branch this will impose an additional delay when the branch is not taken.

- Delayed branch

  The simplest solution of all is to inform the user that we have a *delayed branch*. Each control transfer instruction is followed by a *delay slot*. The instruction in this slot will be executed before the branch is taken. The compiler must either fill the *delay slot* with useful code or with a `NOP` instruction.

Control Hazards

- Branch penalty

	branch taken	*branch not taken*	
Without dedicated adder			
Predict branch not taken	3	0	
With dedicated adder			
Always stall	1	1	
Delayed branch	0-1	0-1	(NOPs)
Predict branch not taken	1	0	

Control Hazards

The SPARC specification was designed for pipeline operation - it includes:

- Single delay slot

  The instruction following `CALL JMPL` is always executed.

  For unconditional branches it is not difficult to make good use of this delay slot.

- Optional predict branch taken

  `Bicc` instructions have an *annul* field.

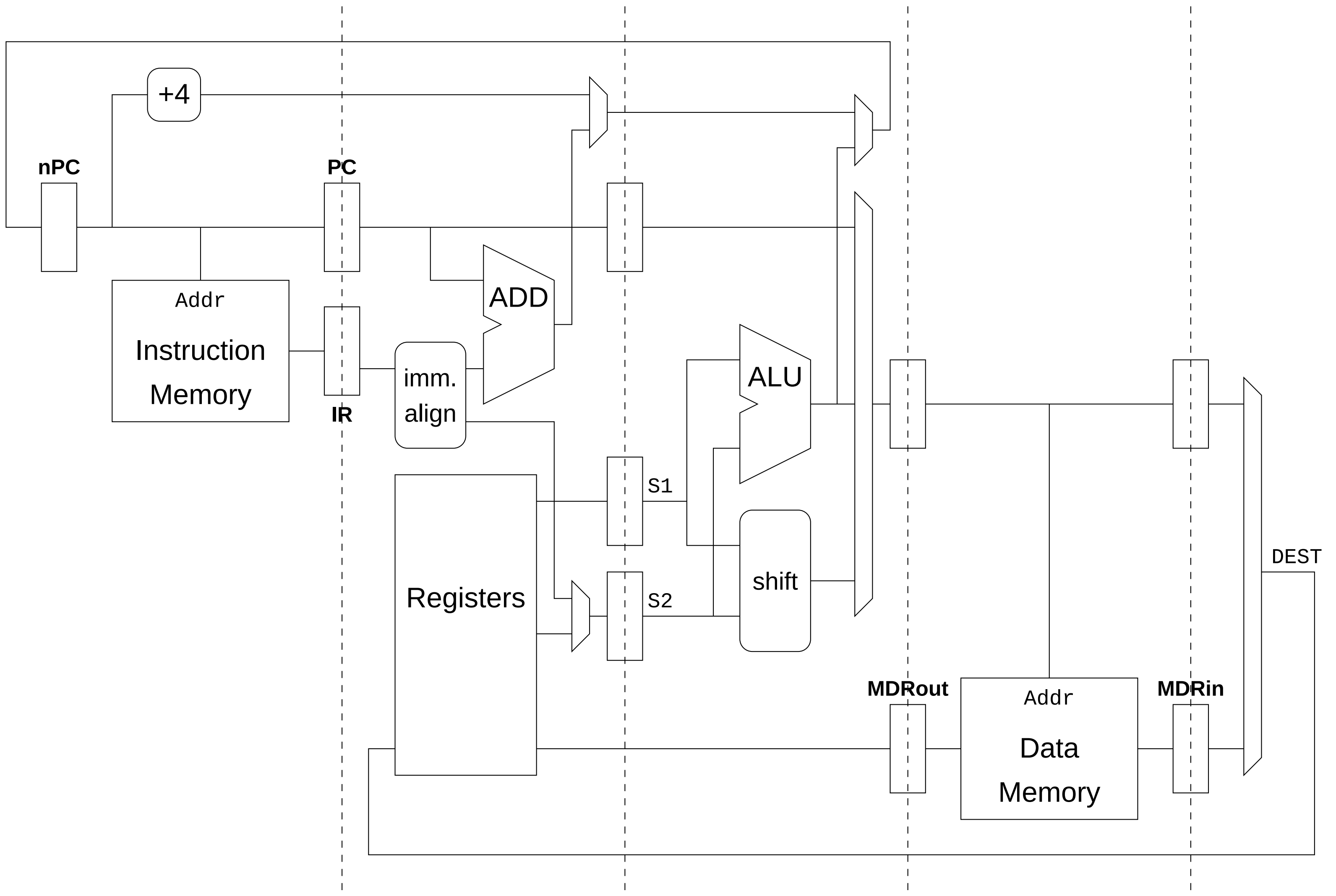
  annul=0 a single delay slot instruction is always executed

  annul=1 the delay slot instruction is flushed where the branch is *not taken*
  this is very efficient for short loops which might otherwise dramatically reduce pipeline performance.

Since register values are not available until the EXE stage our implementation includes:

- Additional single cycle always stall for `JMPL`

SPARC


+4
nPC
PC
Addr
Instruction
Memory
IR
ADD
imm.
align
ALU
S1
S2
shift
Registers
MDRout
Addr
Data
Memory
MDRin
DEST

Control Hazards

- Branch penalties

	delay slot filled		*delay slot empty*	
`CALL`	0		1	
`JMPL`	1		2	
	taken	*not taken*	*taken*	*not taken*
`Bicc`	0	0	1	1
`Bicc,a`	0	1	1	1

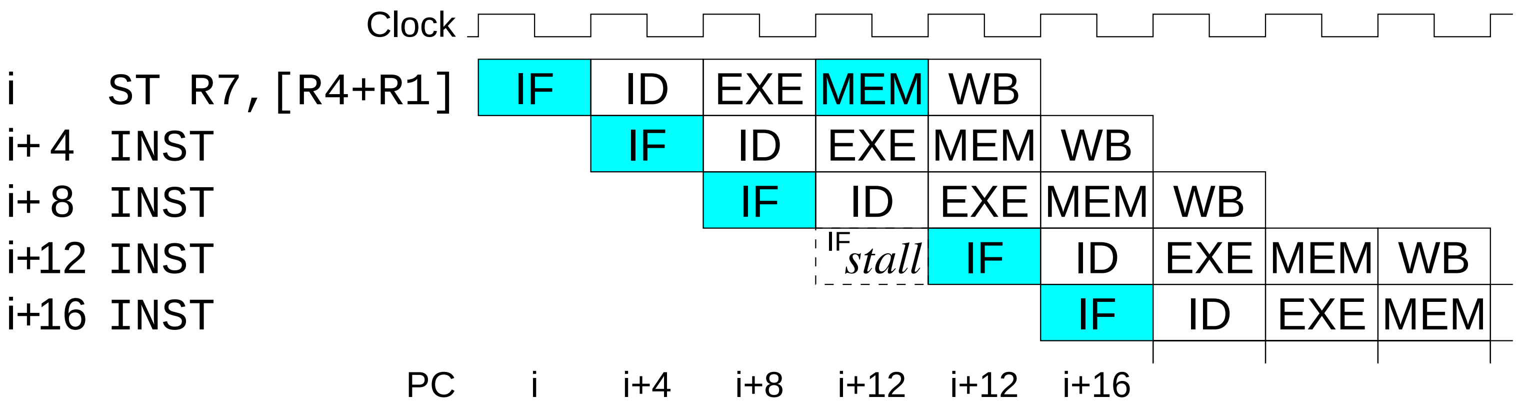
The reduction in processor performance due to control hazards will be dependent on the proportion of executed instructions from the above groups.

The architectural changes we have made have decreased CPI at the expense of complicating the earlier stages of the pipeline. Careful analysis of critical paths would be required to see if the clock frequency must be reduced.

Structural Hazards

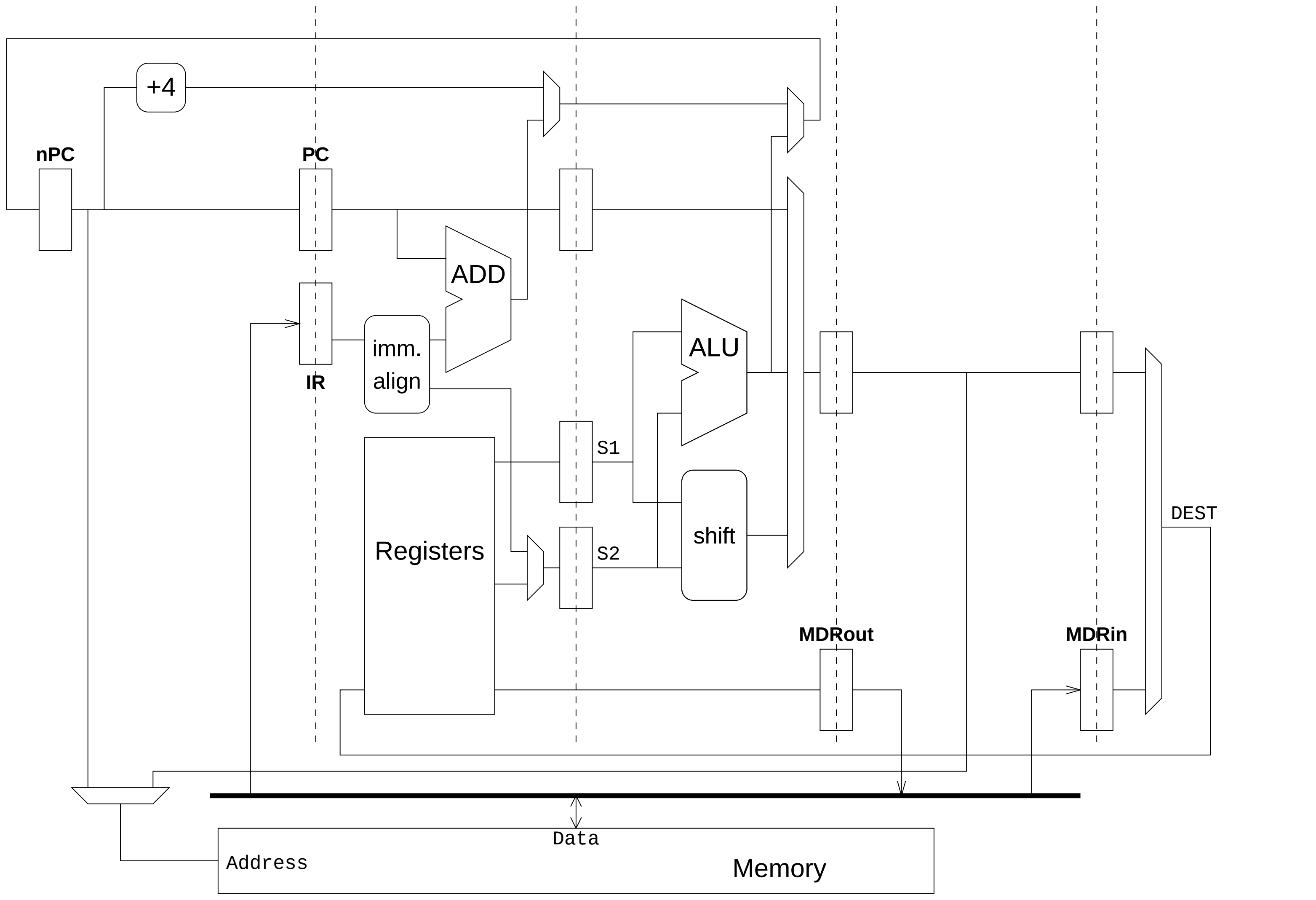
Our Harvard pipeline SPARC has been designed without any possible structural hazards, we can easily illustrate structural hazards by considering a pipeline machine based on the Princeton architecture.

- The memory may be required for both instruction fetch and data access in the same cycle.

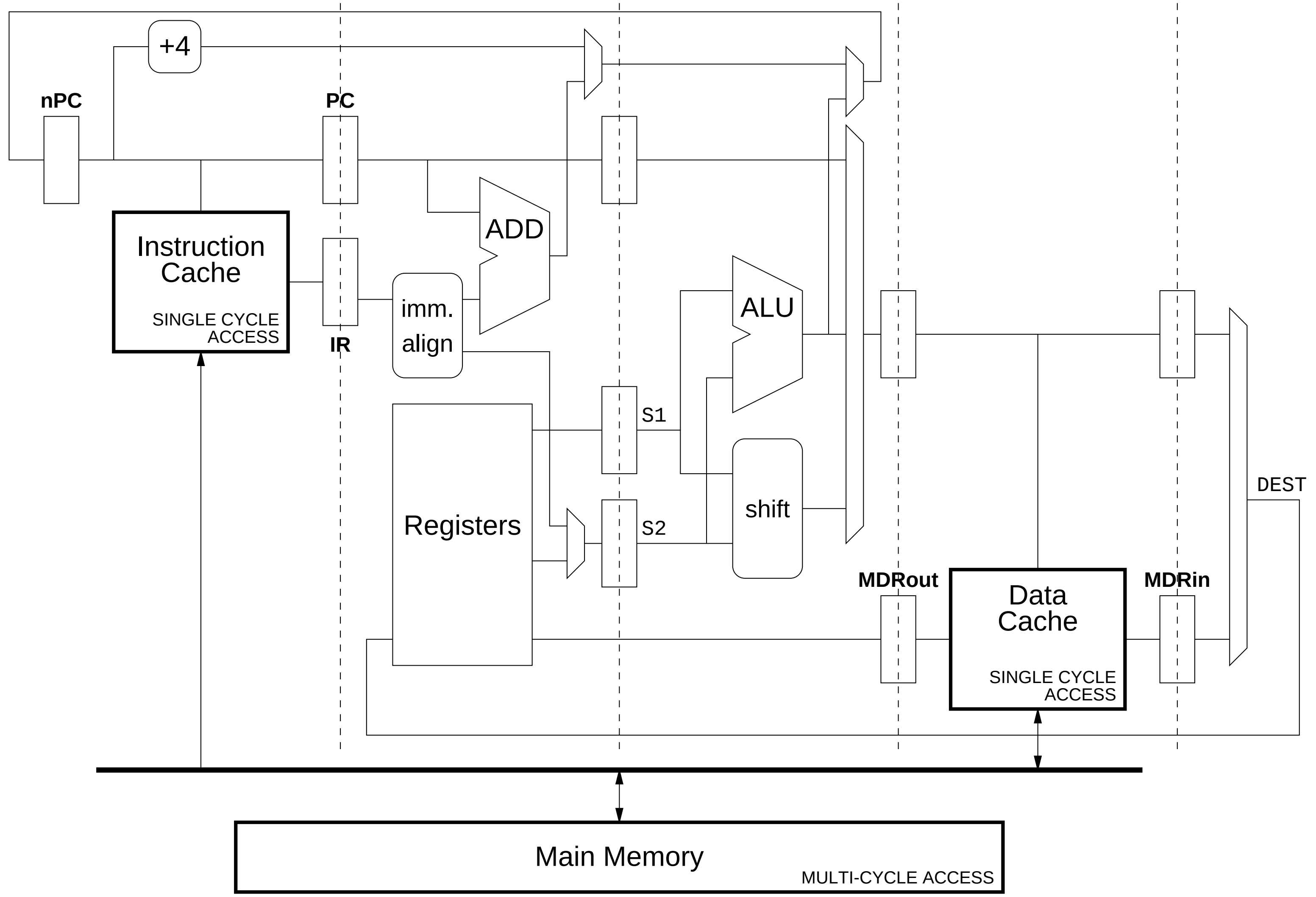


- The data access is given priority and the instruction fetch must be stalled.

Princeton SPARC

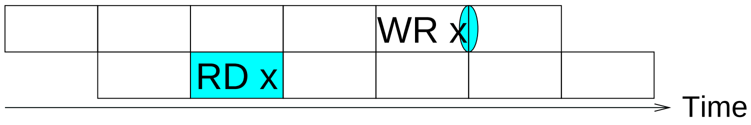


Pseudo-Harvard SPARC



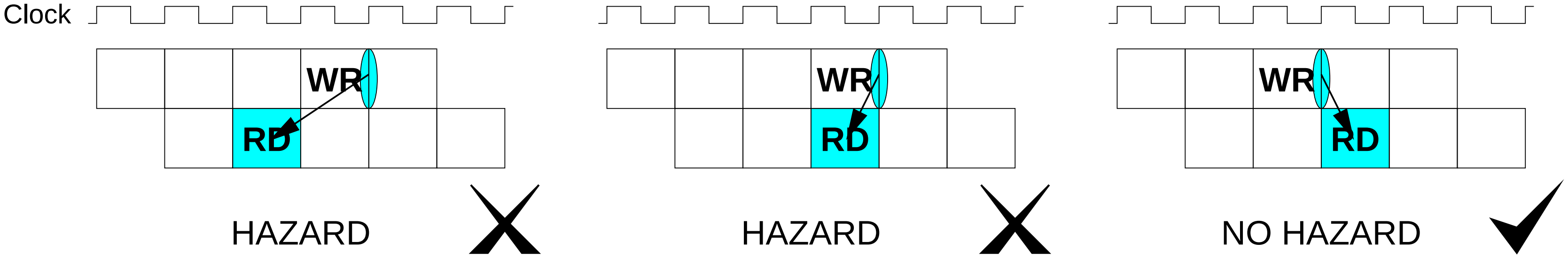
Data Hazards

Data Hazards are of 3 types; RAW, WAR & WAW.



- RAW – *Read after Write*[4]

  A read returns the wrong value since it occurs out of order before a write to the same location.

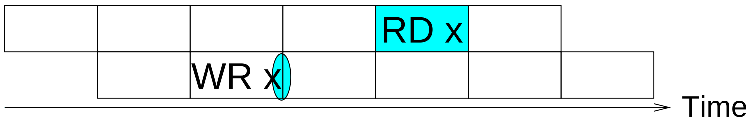


A *Read after Write* hazard is caused by a *True Dependency* since the second instruction *depends* on the result of the first.

[4]note that the name of the hazard indicates the *intended* order or execution rather than the one that gives rise to an error

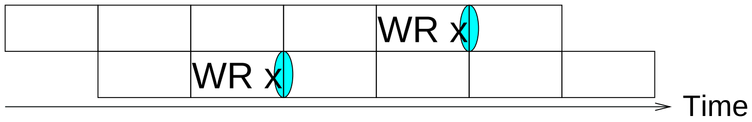
Data Hazards

- WAR – *Write after Read*



  A read returns the wrong value since it was preceded by an out of order write to the same location.

- WAW – *Write after Write*



  A value is incorrectly updated since two writes to the location occur out of order.

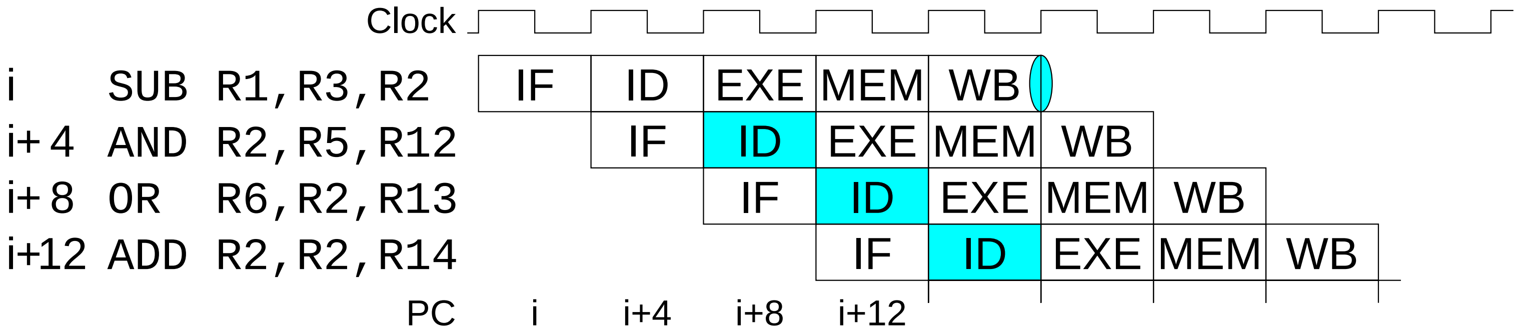
  Write after Read and *Write after Write* hazards are caused by *False Dependencies*. The hazard arises from the re-use of a register rather than any true data dependency.

Since we update registers only at the end of an instruction, our architecture illustrates no *Write after xxxx* hazards. We need only consider occurrences of *Read after Write*.

Data Hazards

- *Read after Write*

  Consider the following set of instructions:



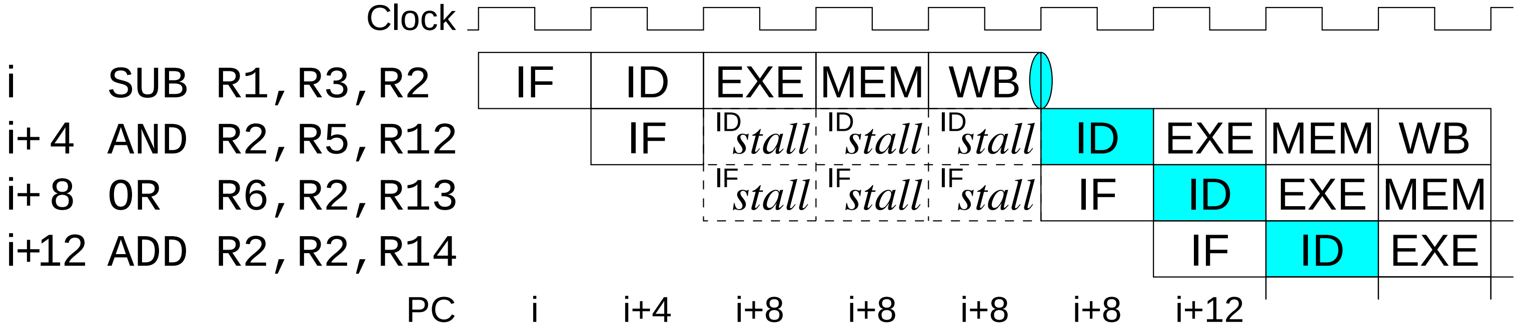
The AND, OR and ADD instructions read R2 before it is written by the SUB instruction. All three instructions will receive the old value of R2.

These RAW hazards are described as *Define-Use* hazards since R2 is defined by the SUB instruction and used by the AND, OR and ADD instructions. We shall consider the more awkward *Load-Use* hazards later.

Data Hazards

- We must *stall* the pipeline until the data is available.

  A *Hazard detection unit* keeps track of out of date register values.

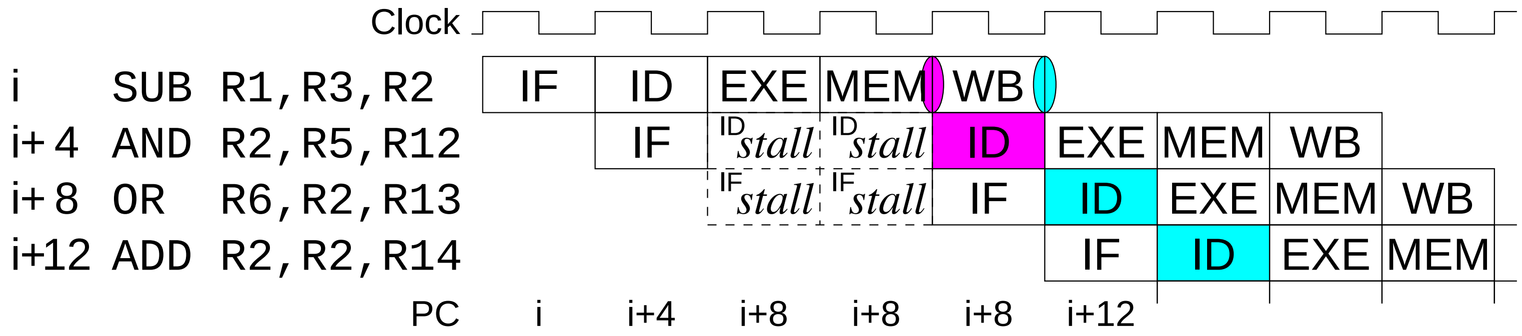


Since we might reasonably expect an instruction to depend on the result of the previous instruction, the impact on our CPI is dramatic.

Data Hazard Workaround

1. Transparent Register File

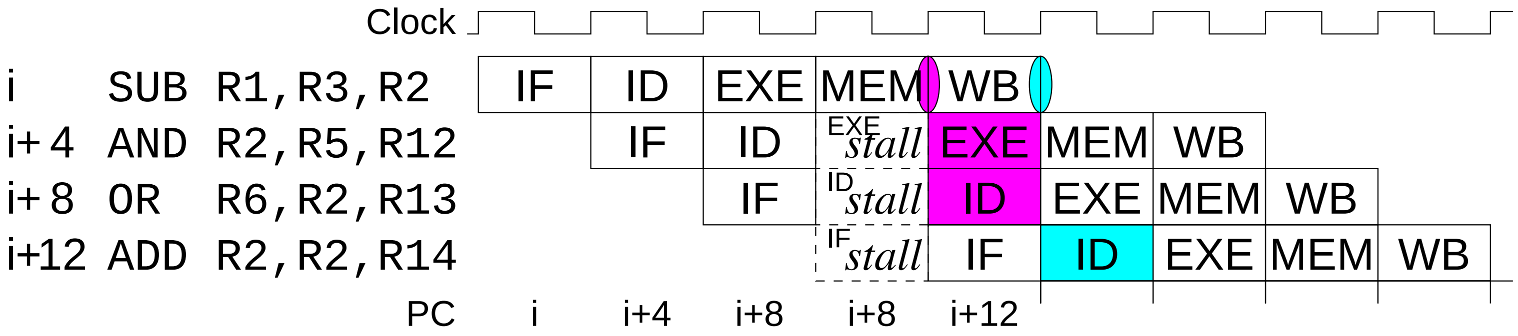
The first step is to note that the result is fed into the register file at the beginning of the WB stage. If we allow the updated register to be transparent then we can feed the result straight from the WB stage to the ID stage.



We have reduced the hazard penalty to two cycles.

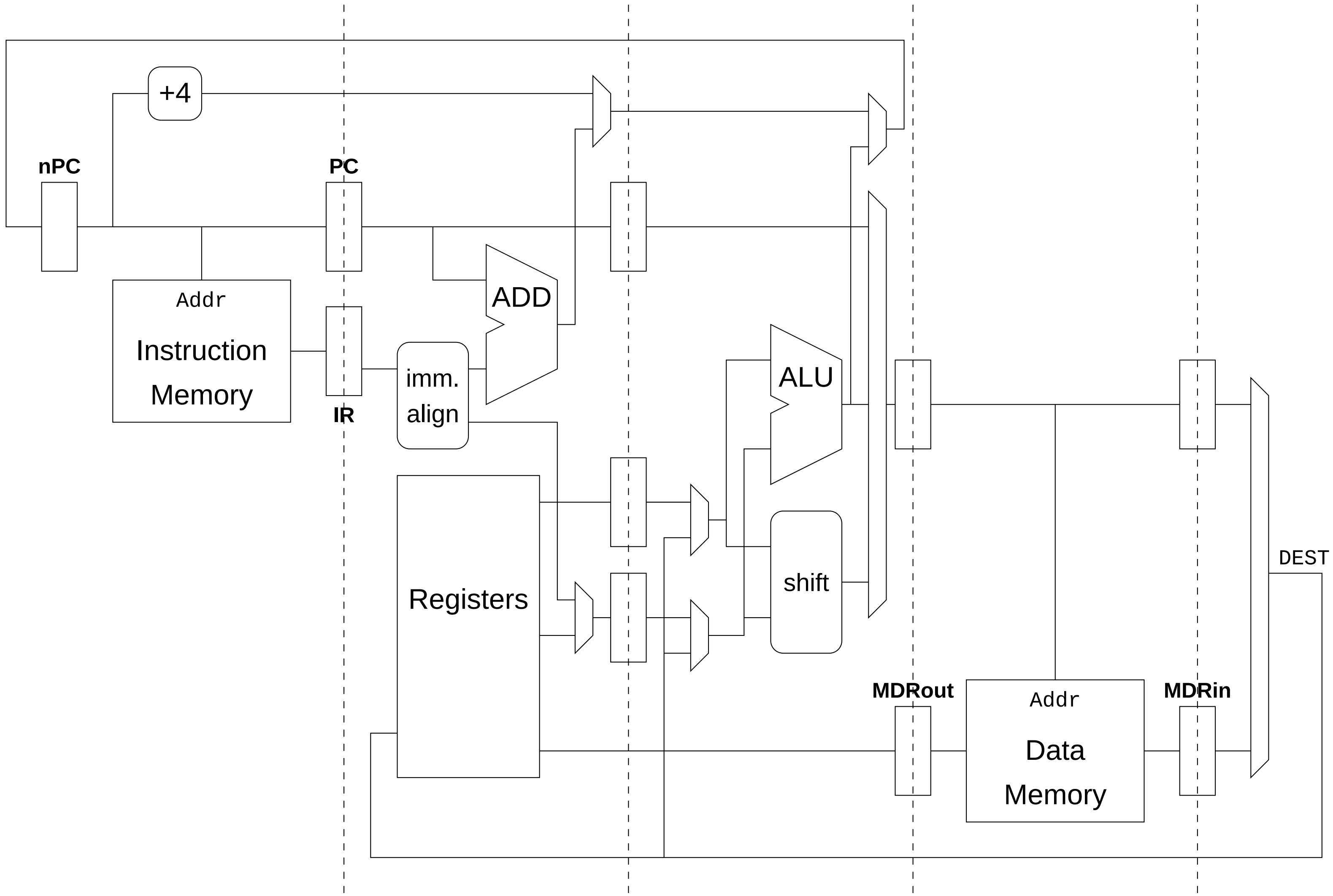
2. Data Forwarding

The second step is to note that the result is not strictly required by the subsequent instruction until the beginning of the execute stage. If we provide an additional datapath which bypasses the register file and the S1 and S2 registers then we can feed the result from the beginning of the WB stage to the beginning of the EXE stage.



We have reduced the hazard penalty to one cycle.

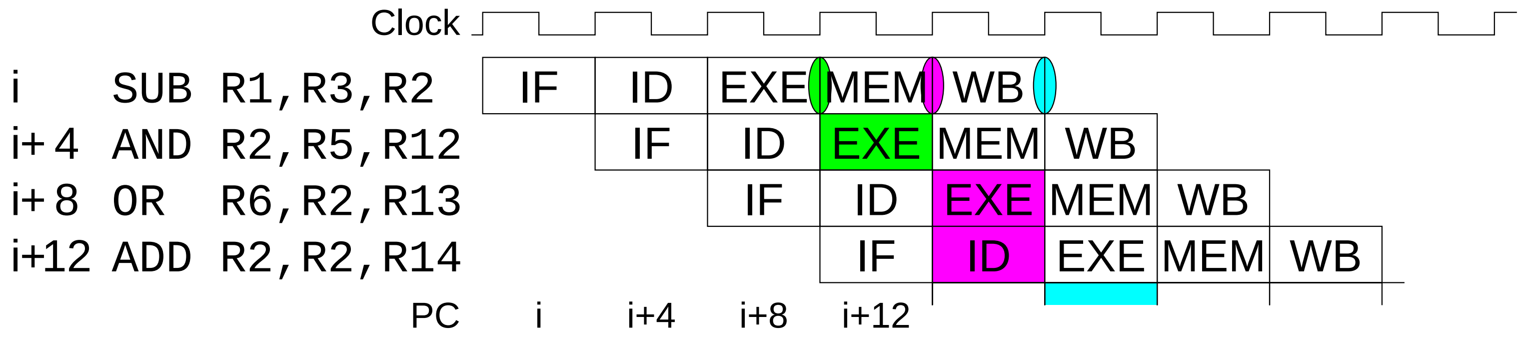
SPARC



+4
nPC
PC
Addr
Instruction
Memory
IR
imm.
align
ADD
ALU
shift
Registers
MDRout
Addr
Data
Memory
MDRin
DEST

3. More Data Forwarding

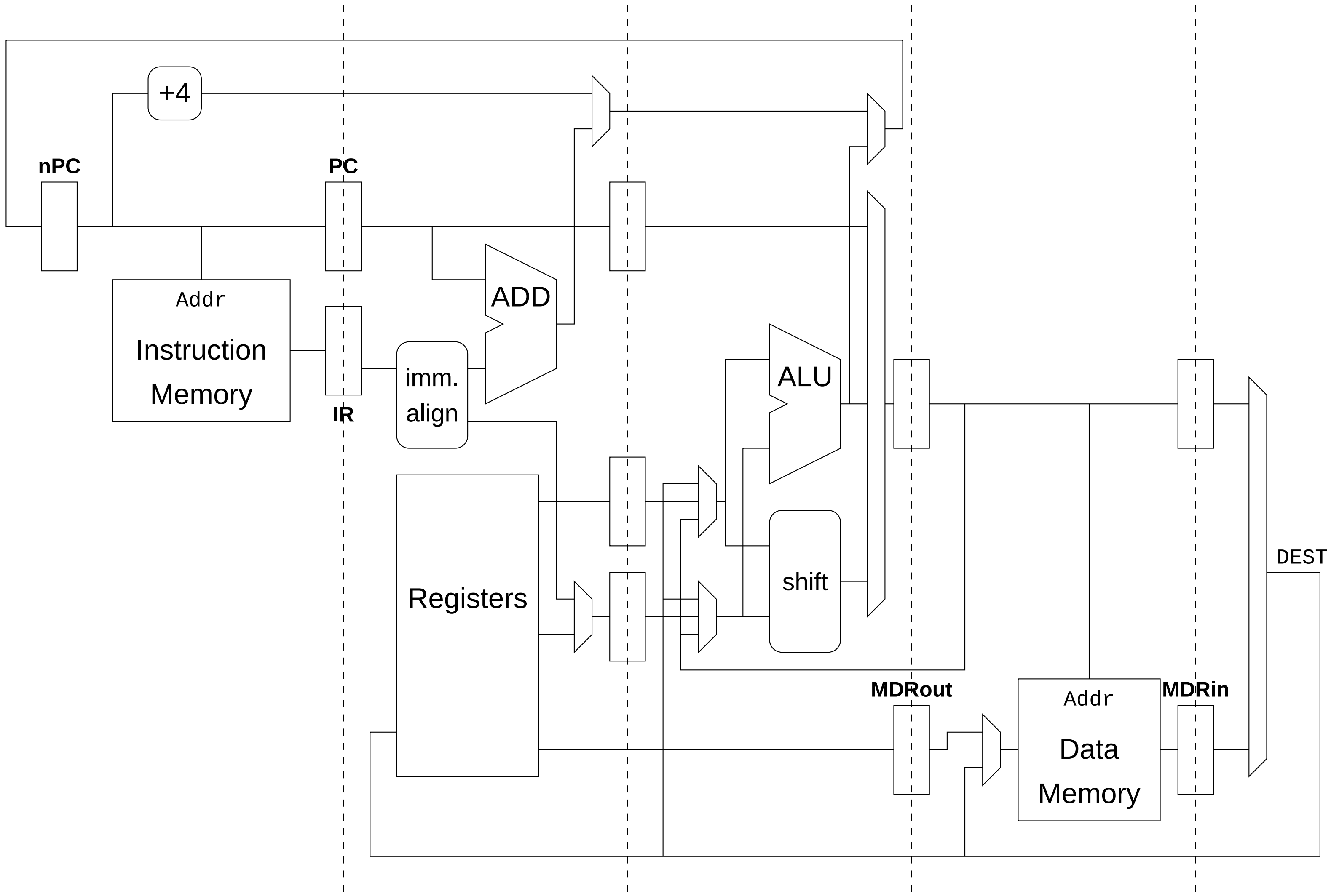
By a similar modification we can forward data from the beginning of the MEM stage to the new S1 and S2 multiplexors at the beginning of the EXE stage[5].



In this case we have no hazard at all.

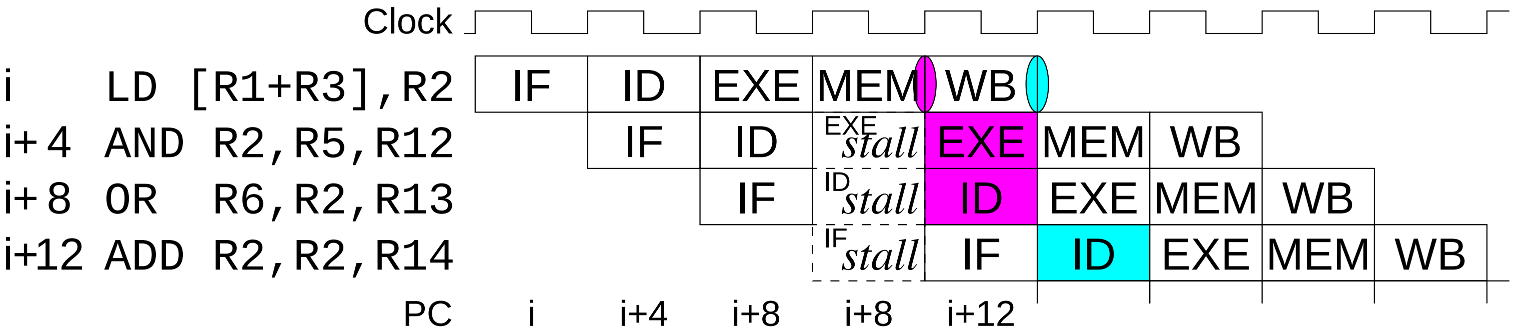
[5] the diagram also shows an additional multiplexor which ensures the integrity of values stored to memory

SPARC


+4
nPC
PC
Addr
Instruction
Memory
IR
imm.
align
ADD
ALU
shift
Registers
DEST
MDRout
Addr
Data
Memory
MDRin

Residual *Load-Use* Data Hazard[6]

A feed forward from the beginning of the MEM stage is not appropriate where the MEM stage is in the critical path. Hence we may still see a data hazard after a `LD` instruction.



Although some machines support a *delayed load*, where the new value is unavailable for the next instruction, the SPARC requires a single cycle stall in order to maintain assembly language semantics[7].

---

[6] we have now removed all the *define-use data hazards*

[7] note that a clever compiler will schedule instructions in order to avoid this hazard

Stalls and Bubbles

Instruction Oriented Pipeline Diagram

In this pipeline diagram, time runs left to right with one row being allocated per instruction.

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10
`LD [R1+20],R2`	IF	ID	EXE	MEM	WB					
`AND R2,R5,R4`		IF	ID	EXE *stall*	EXE	MEM	WB			
`OR  R2,R6,R8`			IF	ID *stall*	ID	EXE	MEM	WB		
`ADD R4,R2,R9`				IF *stall*	IF	ID	EXE	MEM	WB	
`SUB R6,R7,R1`						IF	ID	EXE	MEM	WB

Time

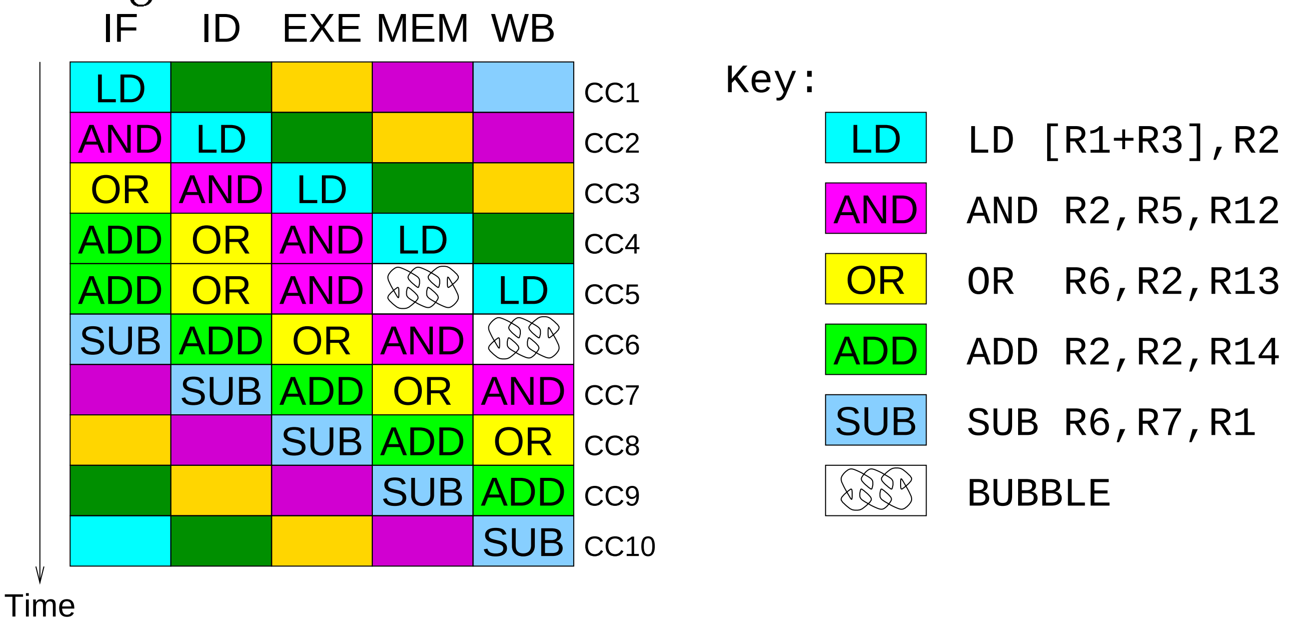
In the 4^{th} cycle the EXE stage is *stalled* in order to avoid a *load-use* hazard. In turn this stalls IF & ID while MEM (& WB) are allowed to continue. The result is a *bubble* in the pipe which appears in the MEM stage in the 5^{th} cycle. It is not possible to show this *bubble* on an instruction oriented pipeline diagram.[8]

[8]Patterson & Hennessy describe bubbles very badly, often using the words stall and bubble interchangeably. Computer Organization & Design (Figure 6.45) attempts to show a bubble on an instruction oriented pipeline diagram - This is wrong!

Stalls and Bubbles

Stage Oriented Pipeline Diagram

In this pipeline diagram, time runs top to bottom with one column being allocated per pipeline stage.



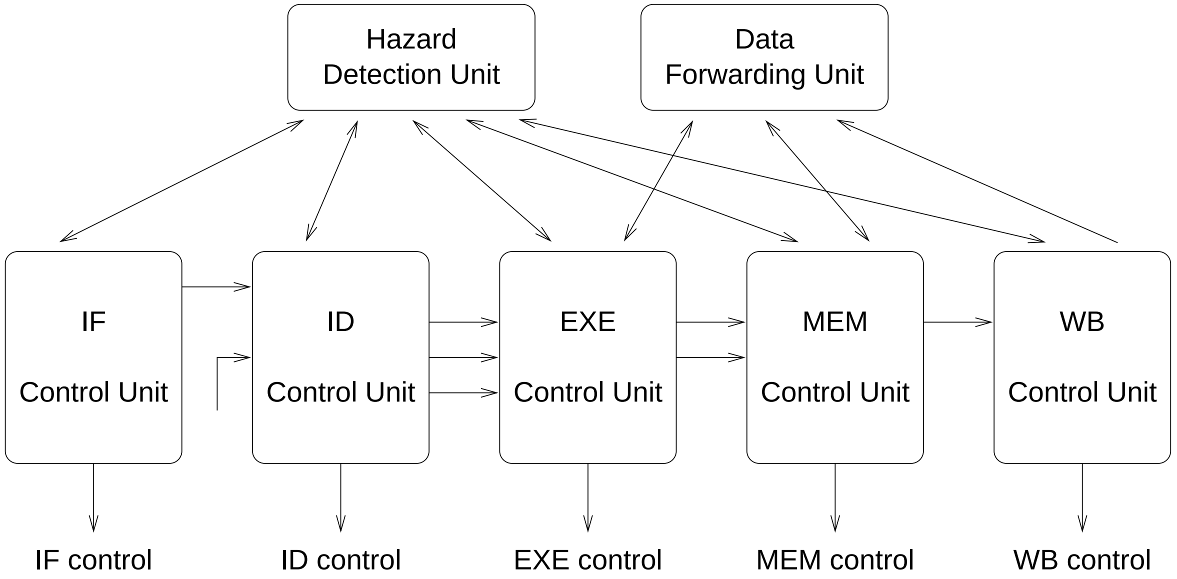
Here it can be seen that the *bubble* is simply an empty pipeline stage which arises when the `AND` instruction is stalled while the `LD` instruction continues. Once created a *bubble* will usually advance through the pipe until it reaches the last stage as if it was a normal instruction. In the 6^{th} cycle we see the *bubble* in the WB stage.

Controller Redesign

- To take account of the pipeline complexities introduced by hazard avoidance, a redesign of the control structure is required.

  We need to re-introduce an element of central control:

  - the hazard detection unit is responsible for pipeline flush and stall
  - the data forwarding unit controls the data forwarding multiplexors



Pipeline Processor Performance

- Let us compare the performance of a pipelined SPARC with a non-pipelined version.
  - Since the instruction sets are identical we can compare them on instruction rate alone.

$$Speedup = \frac{Instruction\ Rate_{pipe}}{Instruction\ Rate_{non\ pipe}}$$

$$Instruction\ Rate\ (Mips) = \frac{Clock\ Frequency\ (MHz)}{Av.\ Cycles\ per\ Instruction}$$

  - Since the number of cycles per instruction is not constant we must consider the frequency of instructions within executed code[9].

[9] note that we are interested in the dynamic occurrence of instructions during execution rather than their static occurrence within code

Pipeline Processor Performance

Frequency of executed instructions for *SPARCjnr* processor[10]:

Register register instructions		
ADD ADDcc SUB SUBcc ADDX ADDXcc SUBX SUBXcc XOR XORcc AND ANDcc ANDN ANDNcc OR ORcc ORN ORNcc XNOR XNORcc		39%
SLL SRL SRA		7%
SETHI		5%
Load store instructions		
LD	(no data hazard)	14%
LD	(with data hazard)	4%
ST		9%
Control transfer instructions		
CALL		3%
JMPL		4%
Bicc(,a)	(taken)	9%
Bicc	(untaken)	3%
Bicc,a	(untaken)	3%

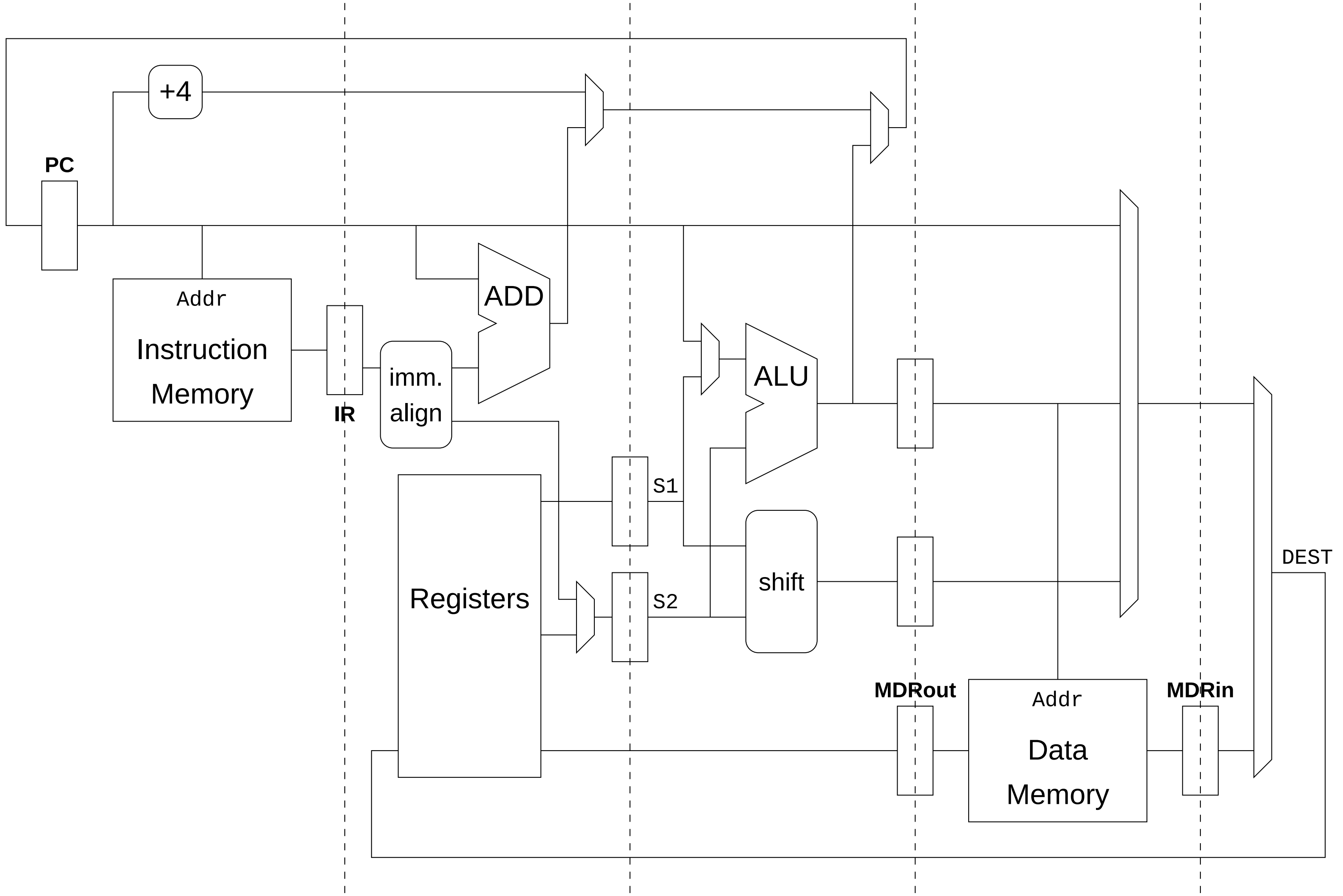
[10]extrapolated and estimated from available data

Pipeline Processor Performance

CPI data for non-pipelined and pipelined *SPARCjnr*:

			non-pipelined	pipelined
ADD ADDcc ...		39%	4	1
SLL SRL SRA		7%	4	1
SETHI		5%	4	1
LD	(no data hazard)	14%	5	1
LD	(with data hazard)	4%	5	2
ST		9%	4	1
CALL		3%	2	1
JMPL		4%	3	2
Bicc(,a)	(taken)	9%	2	1
Bicc	(untaken)	3%	2	1
Bicc,a	(untaken)	3%	2	2
			3.78	1.11

Non-Pipeline Architecture for Comparison



Pipeline Processor Performance

The instruction rate is calculated using a weighted average of the CPI values over a set of real programs or benchmarks.

$$\begin{aligned} Av.\,Cycles\;per\;Instruction_{non\;pipe} &= 4\times 39\% + 4\times 7\% + 4\times 5\% + 5\times 14\% + \\ &\quad 5\times 4\% + 4\times 9\% + 2\times 3\% + 3\times 4\% + \\ &\quad 2\times 9\% + 2\times 3\% + 2\times 3\% \\ &= 3.78\,CPI \end{aligned}$$

We have assumed that the non-pipe architecture does not waste cycles[11].

$$\begin{aligned} Av.\,Cycles\;per\;Instruction_{pipe} &= 1\times 39\% + 1\times 7\% + 1\times 5\% + 1\times 14\% + \\ &\quad 2\times 4\% + 1\times 9\% + 1\times 3\% + 2\times 4\% + \\ &\quad 1\times 9\% + 1\times 3\% + 2\times 3\% \\ &= 1.11\,CPI \end{aligned}$$

This value is for a pure Harvard implementation. A Princeton implementation would have a greater CPI due to structural hazards.

[11] `ADD` completes within 4 cycles since it does not require a MEM cycle

Pipeline Processor Performance

- Clock Frequency

  For the purposes of this exercise we shall assume that the clock frequencies of the two machines are the same. In fact we might expect the non-pipelined implementation to support a faster clock since it doesn't have the overheads of the hazard avoidance circuitry.

- Speedup

$$Speedup = \frac{Clock\ Frequency_{pipe}}{Av.\ Cycles\ per\ Instruction_{pipe}} \div \frac{Clock\ Frequency_{non\ pipe}}{Av.\ Cycles\ per\ Instruction_{non\ pipe}}$$

$$= \frac{Av.\ Cycles\ per\ Instruction_{non\ pipe}}{Av.\ Cycles\ per\ Instruction_{pipe}}$$

$$= \frac{3.78\ CPI}{1.11\ CPI} = 3.4$$

Pipeline Processor Performance

- Architecture enhancements

  Each time an enhancement is considered it must be remembered that its effect will be limited by the frequency with which enhanced instructions occur in real code.

- *Amdahl's Law* states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.

$$Execution\ time_{new} = Execution\ time_{old} \times (Fraction_{unenhanced} + \frac{Fraction_{enhanced}}{Speedup_{enhanced}})$$

$$\begin{aligned} Speedup &= \frac{Execution\ time_{old}}{Execution\ time_{new}} \\ &= \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}} \end{aligned}$$

Amdahl's Law - example

- The addition of a `MULScc` instruction to our *SPARCjnr* architecture will speedup multiplication performance by a factor of 4.
  - Integer multiply occupies 30% of processor time for vision processing application:

$$Speedup_{with\ MULScc} = \frac{1}{70\% + \frac{30\%}{4}} = 1.29$$

  - Integer multiply occupies 1% of processor time for wordprocessing program:

$$Speedup_{with\ MULScc} = \frac{1}{99\% + \frac{1\%}{4}} = 1.008$$

This analysis helps us to evaluate the cost/benefit of an enhancement. It may also rule out some enhancements altogether – consider a `MULScc` instruction which increases multiply performance by 4 but causes a 2% increase in clock period.

Pipeline Processor Performance

- When designing processor hardware a useful rule is:

  Make the Common Case Fast

  This is the philosophy of RISC computers.

  - We have invested significant effort in the speeding up of conditional branch instructions since they are common (16% of all executed instructions)

  - We have provided no specialist hardware for integer division since it is required very rarely.

  Amdahl's Law helps us to quantify this simple rule.

The Compiler

- Create functional code

  The compiler must understand the semantics of the available instructions

- Create fast code

  The compiler must understand the speed implications of using different instructions.

  For pipeline architectures the compiler should optimize code to avoid:

  - pipeline stall
  - pipeline flush
  - NOP (in load or branch delay slot)

  In all cases instructions are reordered (scheduled) to create the most efficient code.

Pipeline Processor Performance

Scheduling for fast code on *SPARCjnr*

- schedule useful branch delay slot instructions
  consider the impact on performance if all delay slots are `NOP`:

$$\begin{aligned}
Av.\ Cycles\ per\ Instruction_{pipe} &= 1 \times 39\% + 1 \times 7\% + 1 \times 5\% + 1 \times 14\% + \\
&\quad 2 \times 4\% + 1 \times 9\% + 2 \times 3\% + 3 \times 4\% + \\
&\quad 2 \times 9\% + 2 \times 3\% + 2 \times 3\% \\
&= 1.30\ CPI
\end{aligned}$$

- avoid data hazard after LD
- optimize for conditional branch taken
  where *annul* is used a taken branch is faster than an untaken one.

When writing a compiler for our processor we wish to:

Make the Fast Case Common