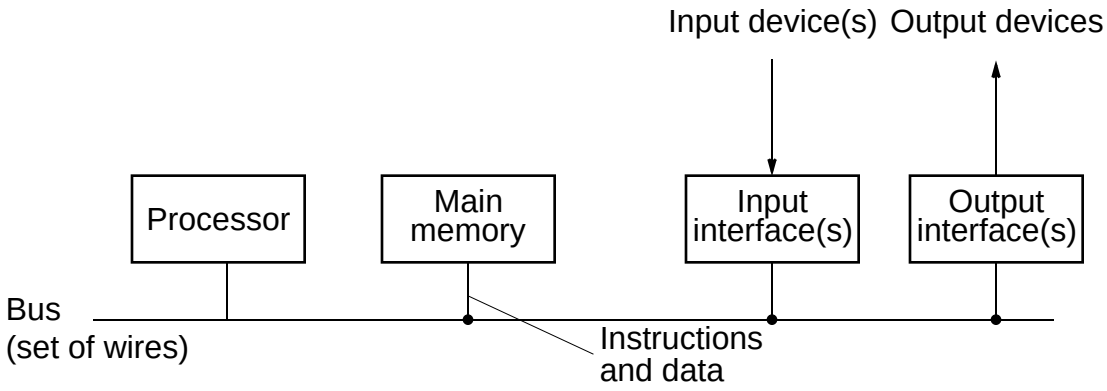
Lecture 2

Internal Structure of Computer

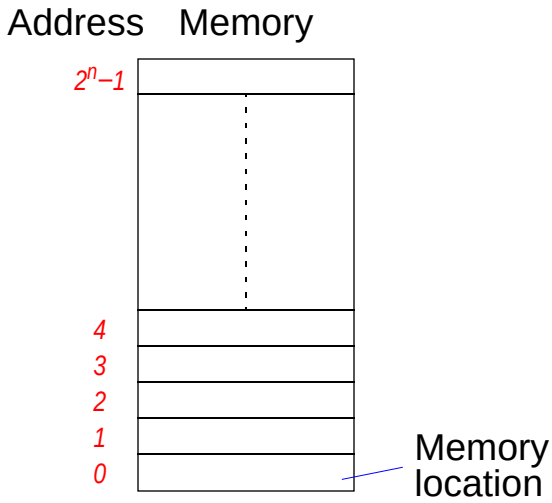
Usually processor, main memory and I/O interfaces connect through central "bus", a collection of wires connecting all major components through which information is transferred.



First used on minicomputers in 1970s (PDP 8 and PDP 11) and subsequently on microprocessors and all present-day systems (with variations e.g. dedicated buses).

Main memory

Set of storage locations. Any location can be accessed at high speed in any order (random access memory). Each location given a unique address (a binary number starting from zero). Each "addressable" location holds fixed number of bits (binary digits) - normally 8 bits (a byte). WHY?



2^n locations require n-bit address

From ITCS 3182

Slide 33

Memory used to hold machine instructions and data.

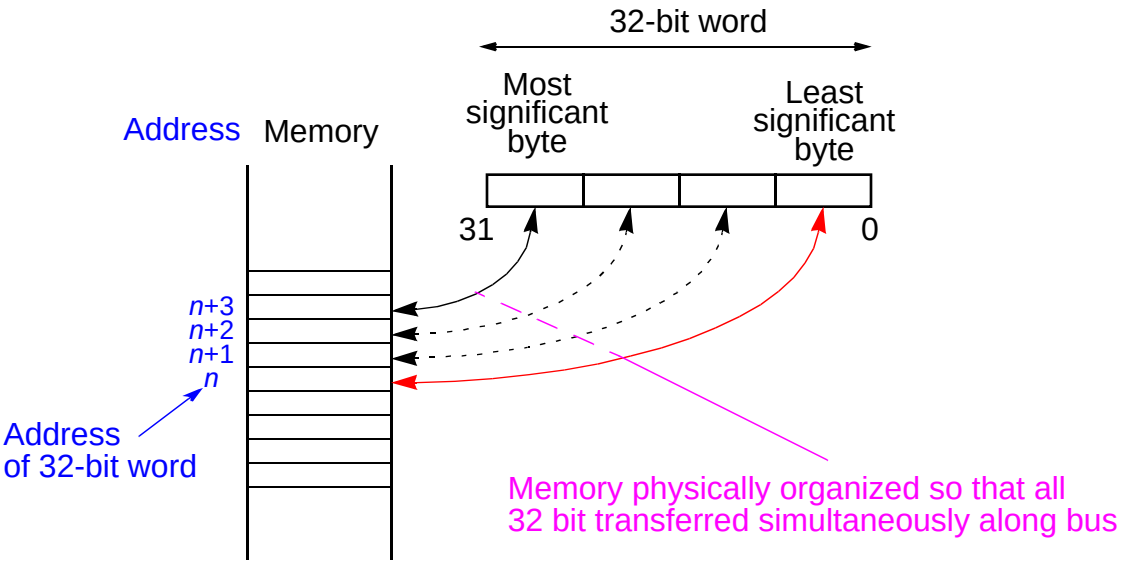
If more than 8 bits needed, consecutive locations used.

Then address given by address of first location.

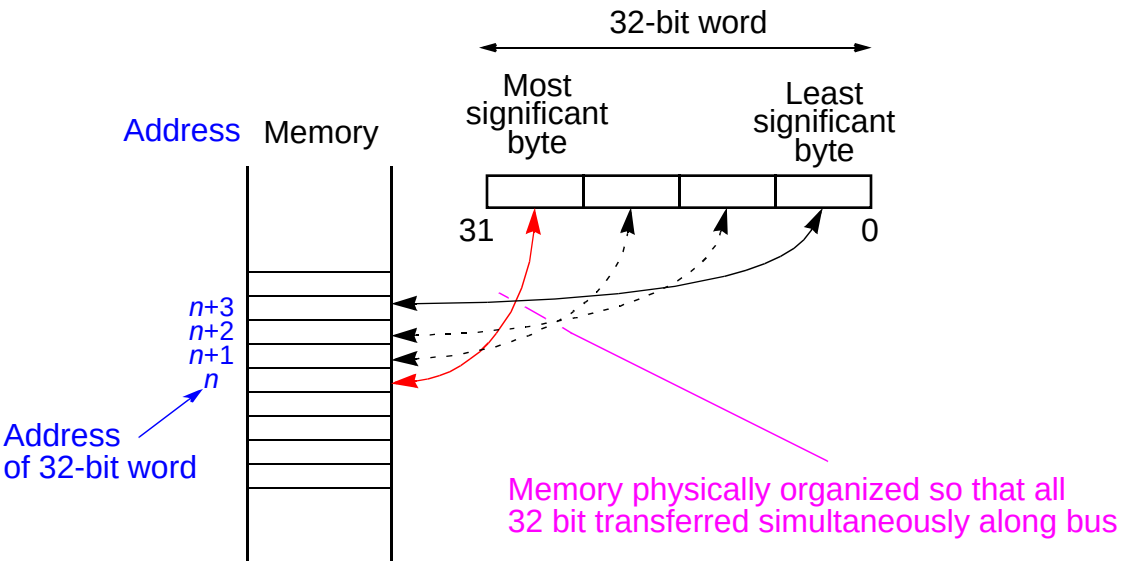
First location can hold least significant or most significant byte depending upon convention of processor:

Slide 34

Little endian (little end first)



Big endian (big end first)

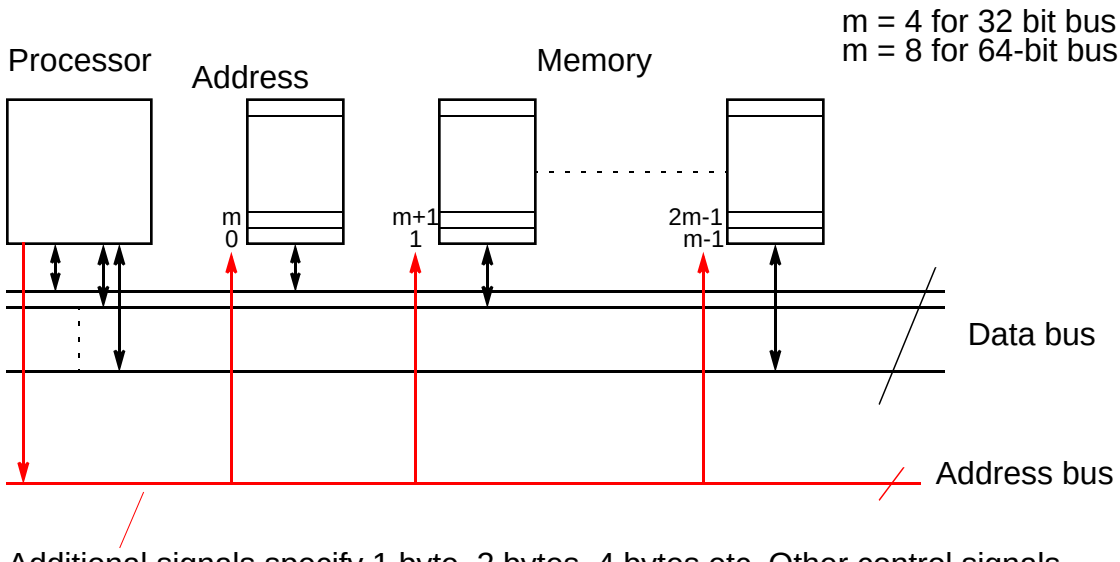


Intel uses little endian

Mac/SUN use big endian

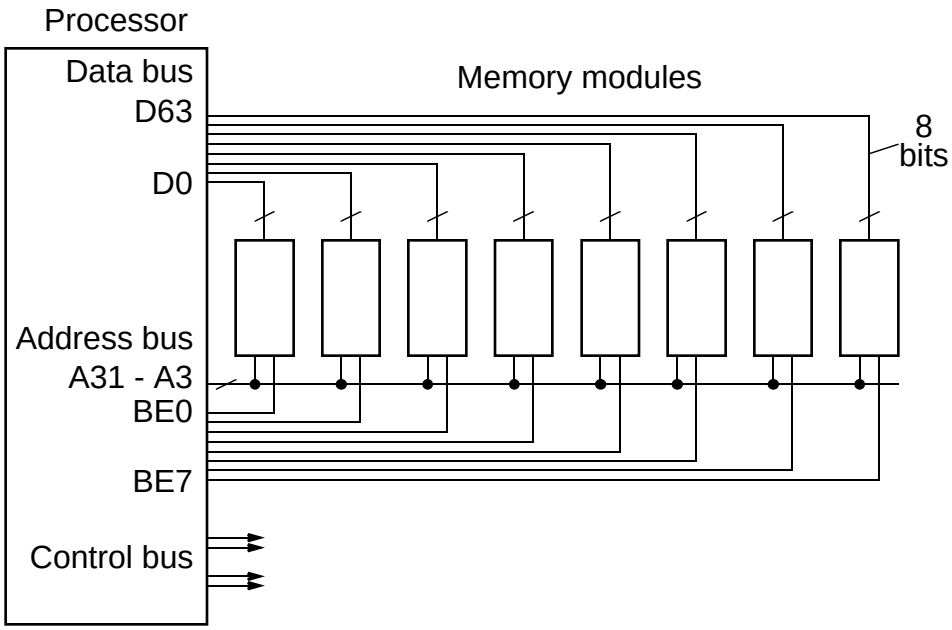
Data path between processor and memory

Normal more than 8 bits transferred simultaneously between processor and main memory - typically 32 bits or 64 bits for modest performance systems:



Additional signals specify 1 byte, 2 bytes, 4 bytes etc. Other control signals.

Example of a 64-bit processor bus (Pentium)



Machine Instructions

Binary encoded instructions that processor will execute. Held in memory.

Various possible formats.

First part of instruction typically specifies operation (add, subtract etc.) - the so-called op-code.

Op-code	Identifies operands and result location

Rest of instruction specifies the locations of the numbers (operands) to be used in operation and where result is to be stored - if an operation that uses stored numbers and produces a numeric result - some operations alter the instruction sequence or produce other effects.

Slide 39

Op-code encoding

Suppose 100 different operations, add subtract, multiply, divide, ...
7 bits sufficient ($2^6 < 100 \leq 2^7$). Could allocate one pattern for each operation:

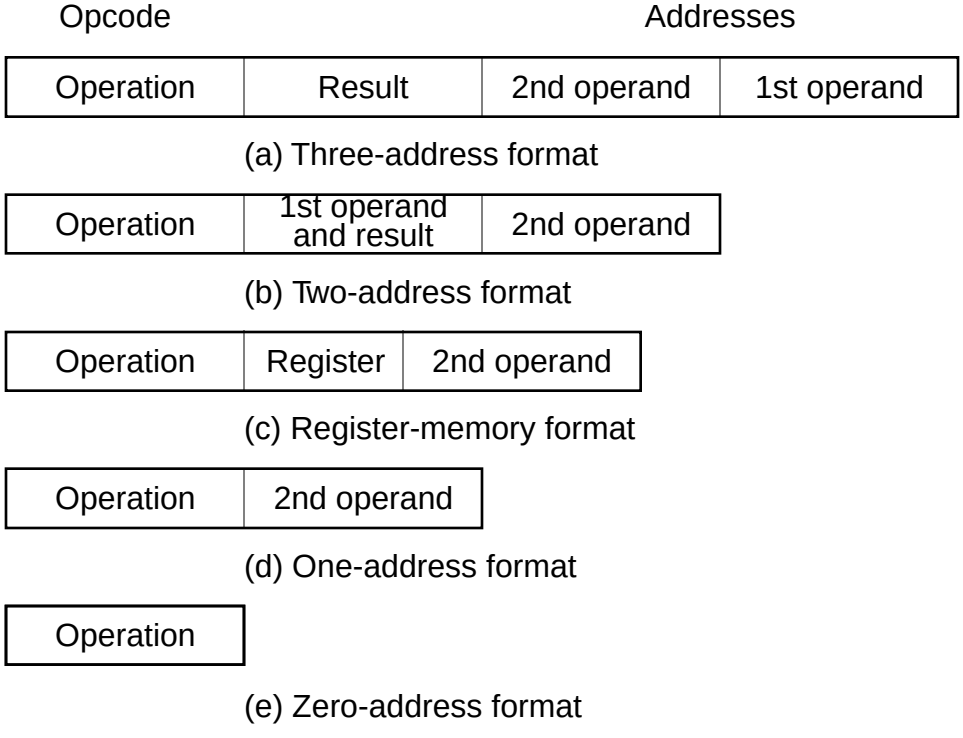
	op-code
ADD (“ADD”)	0000000
SUBTRACT (“SUB”)	0000001
MULTIPLY (“MUL”)	0000010
DIVIDE (“DIV”)	0000011
. .	

Sometimes more complex encoding used, e.g. first bits specify class of operation and remainder of op-code specifies operation within class.
Many possibilities.

Slide 40

Instruction Formats

Basic way of identifying operands is by their addresses. Possible formats:



Slide 41

Identifying the operands and result location

Various methods, known generally as the addressing modes.

Principal addressing modes

Absolute (direct) addressing	Operand in memory and memory address of the operand is held in instruction - as in previous slide.
Immediate Addressing	Operand held in the instruction.
Register Direct Addressing	Operand held in register specified in instruction.
Register Indirect Addressing	Operand held in memory. Address of operand location held in register specified in instruction.

Slide 42

Register Indirect Addressing plus Displacement

Similar to register indirect addressing except an offset held in instruction added to register to form effective address.

Implied Addressing — Some operations have implicit location for operand. Its address not specified.

PC relative addressing - used with instructions that can alter execution sequence.

Slide 43

Processor

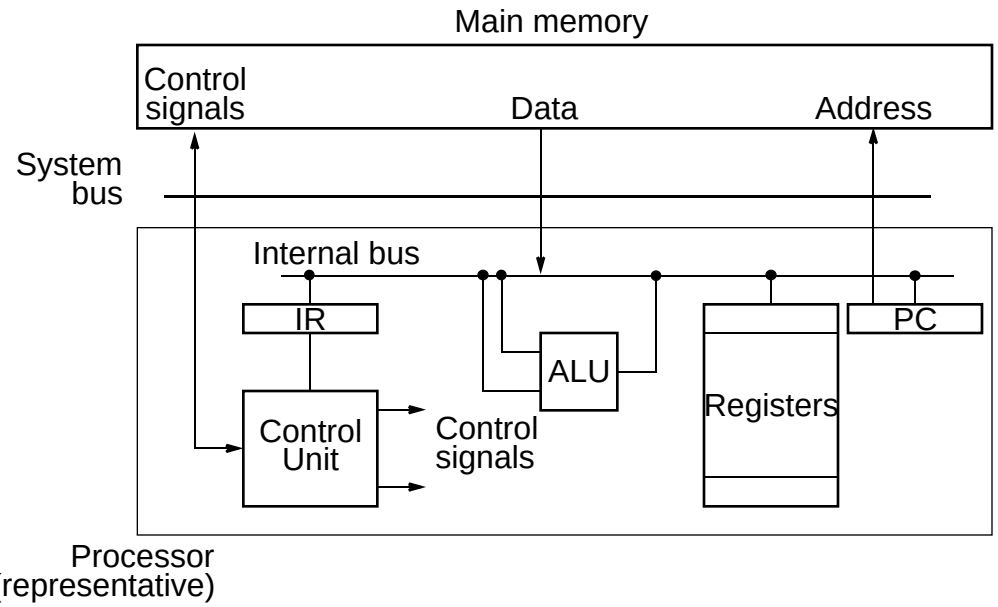
Reads (fetches) the machine instructions (of the executable program) from memory and performance actions specified (executes) them).

Each machine instruction will specify usually a simply operation such as addition, and identifies the numbers to be used.

Slide 44

A very simple processor

(Representative of an early microprocessor - not representative of a modern processor)



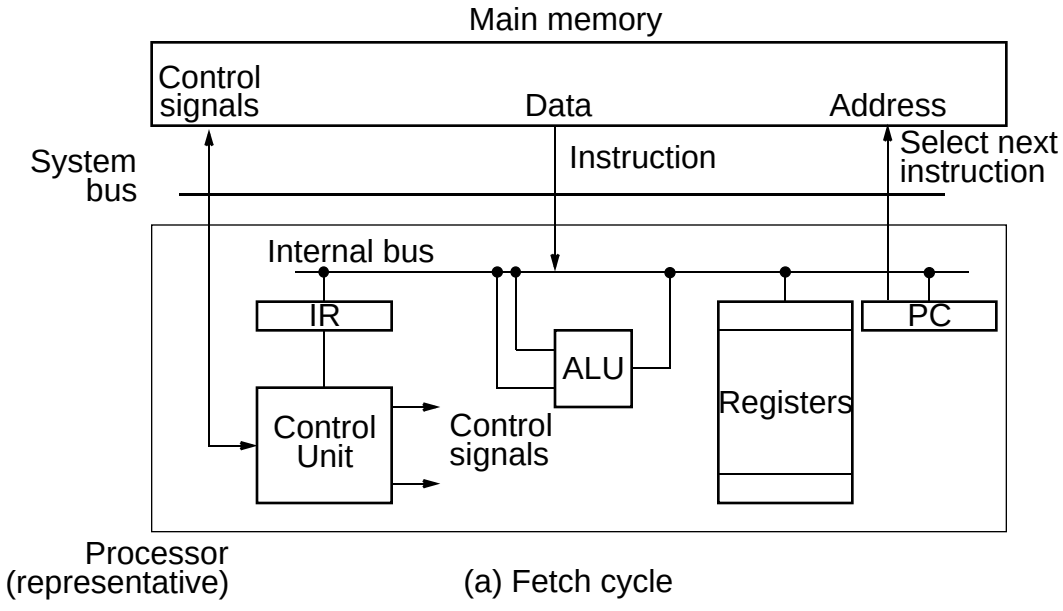
Slide 45

A very simple processor will operate in two phases, a fetch cycle to fetch an instruction and an execute cycle to execute the fetched instruction. These two cycles are repeated as the program is executed.

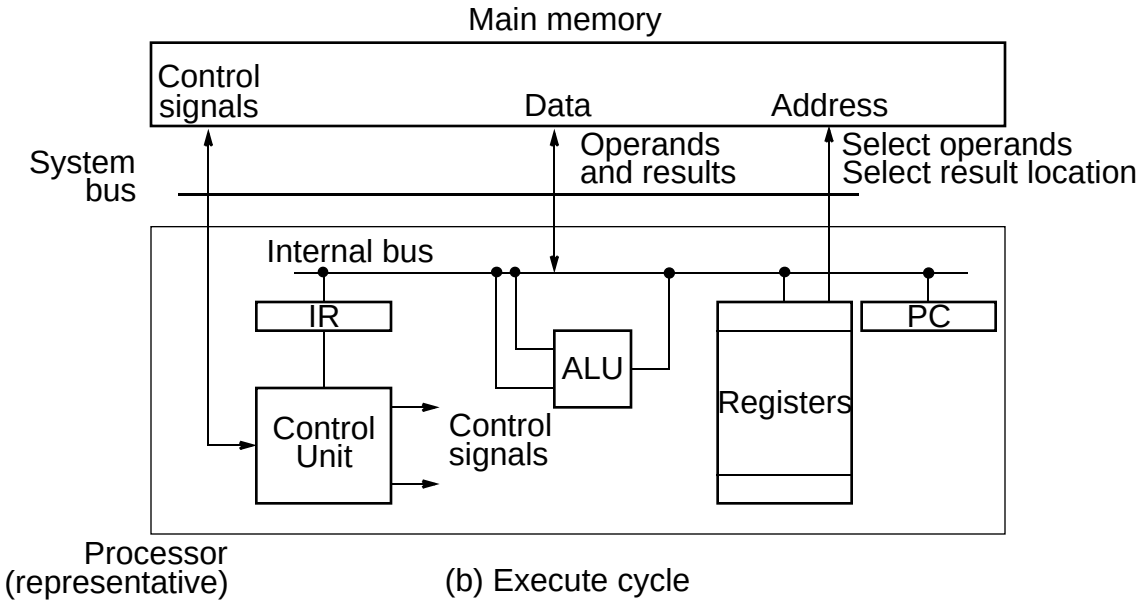
Usually however processors will attempt to fetch the next instruction before the previous one has been fully executed.

Really advanced processors (i.e. modern processors) may fetch multiple instructions simultaneously and attempt to execute more than one simultaneously.

Slide 46



(a) Fetch cycle



Characterizing Performance

MIPs

Traditional system figure of merit is MIPS (millions of instructions per second), defined as:

$$\text{MIPs} = \frac{\text{Number of Instructions in Program}}{\text{Program Execution Time} \times 10^6}$$

Slide 49

MFLOPs

The figure of merit, MFLOPS, (millions of floating point operations per second) is defined as:

$$\text{MFLOPs} = \frac{\text{Number of Floating Point Instructions in Program}}{\text{Program Execution Time} \times 10^6}$$

High performance processors may have very high MFLOP performance i.e. thousands of MFLOPS, called gigaflops, GFLOPS.

Various benchmark programs exist with representative mixes of instructions, for example, SPECint92 and SPECfp92 UNIX benchmarks.

Slide 50

Clock cycles per instruction (CPI)

Clock cycles per instruction (CPI) is defined as:

$$\text{CPI} = \frac{\text{Program execution time (in clock cycles)}}{\text{Number of instructions in program}}$$

CPI is independent of the clock frequency

Can be used to compare different processor designs.

CPI can be less than one if processor capable of executing more than one instruction simultaneously (as most processors can since mid 1990's).

Improving performance

1. Improvements in technology.
2. Software development.
3. Architectural enhancements.

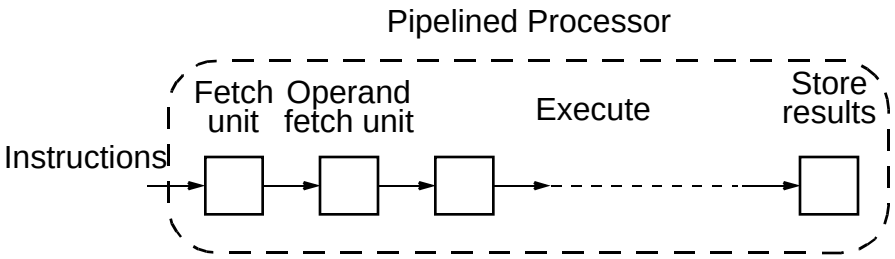
Pipelined processor design

Basic techniques to improve performance - always applied in high performance systems. Operation of processor divided into number of steps, e.g.:

1. Fetch instruction.
2. Fetch operands.
3. Execute operation.
4. Store results

or more steps.

Slide 53



Slide 54

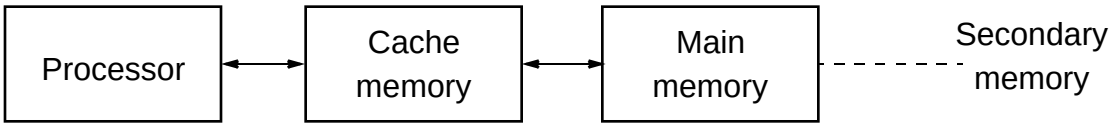
Memory hierarchy

Memory organized in levels of decreasing speed but decreasing cost/bit:

Registers	Storage locations within processor
Cache memory	High speed memory close to processor
Main memory	Random access memory
Secondary memory	not random access but not volatile. Usually being based upon magnetic technology.
Magnetic disk memory	operates several orders of magnitude slower than the main memory. Whereas main memory access time is in order of 20–100 ns, access time on a disk is in range 5–20 ms. Difficult to improve substantially. Gradual improvement over the years.
Virtual memory	Method of hiding the memory hierarchy.

Cache

High speed memory introduced between the processor and main memory:



Lecture 3

Instruction Set Design

The instructions that a processor can execute.

Complex Instruction Set Computers (CISC)

Early computers by necessity had small instruction sets. During the development of computers in the 1960's and 1970's, trend to add instructions to the instruction set sometimes for special purposes (say to help the operating system) leading to sometimes very large number of instructions and addressing modes in the instruction set. Processor instruction sets became every complex. Idea was that better to do in hardware if possible rather than in software.

Slide 57

Reduced instruction set computer (RISC)

Following issues originally lead to RISC concept:

1. Effect of the inclusion of complex instructions.
2. Best use of transistors in VLSI implementation.
3. Overhead of microcode.
4. Use of compilers.

Slide 58

The basic question asked:

"Do the extra instructions indeed increase the speed of the system?"

Inclusion of complex instructions

The inclusion of complex instructions is key issue.

Even if adding complex instructions only added one extra level of gates to a ten-level basic machine cycle, whole CPU slowed down by 10 per cent.

The frequency and performance improvement of the complex functions must first overcome this 10 per cent degradation and then justify the additional cost.

Use of transistors

Trade-off between size/complexity and speed. Greater VLSI complexity leads directly to decreased component speeds.

With increasing circuit densities, a decision has to be made on best way to utilize circuit area.

Is it to add complex instructions at risk of decreasing speed of other operations, or should the extra space on the chip be used for other purposes, such as a larger number of processor registers, caches or additional execution units, which can be performed simultaneously with the main processor functions?

Slide 61

Example

DEC found 20% of VAX instructions required 60% of microcode but were only used 0.2% of the time. Led to micro VAX-32 having slightly reduced set of full VAX instruction set (96 per cent) but very significant reduction in complexity.

Slide 62

Microcode

Factor leading to RISC concept was changing memory technology. CISCs often rely heavily on microprogramming (microcode) - fast control memory inside the processor holds microinstructions specifying the steps to perform for each machine instruction. Microprogramming first used when main memory based upon magnetic core stores and faster read-only control memory could be provided. With move to semiconductor memory, gap between achievable speed of main memory and control memory narrows.

Now, considerable overhead can appear in a microprogrammed control unit, especially for simple machine instructions.

Slide 63

Compilers

There is increased prospect for designing optimizing compilers with fewer instructions.

Difficult for compiler to identify situations where complex instructions can be used effectively.

Key part of RISC development is the provision for an optimizing compiler which can take over some of the complexities from hardware and make best use of registers.

Slide 64

RISC examples

IBM 801

Designed 1975–79 and publicly reported in 1982. Establishes many of features for subsequent RISC designs: Three-register instruction format, with register-to-register arithmetic/logical operations. Only memory operations are to load a register from memory and to store the contents of a register in memory.

IBM 801

All instructions have 32 bits with regular instruction formats.

Programming features include:

- 32 general purpose registers.
- 120 32-bit instructions.
- Two addressing modes: base plus index; base plus immediate.
- Optimizing compiler.

Four-stage pipeline: instruction fetch; register read or address calculation; ALU operation; register write.

Early university research prototypes RISC I/II and MIPS

RISC project –University of California at Berkeley

MIPS (Microprocessor without Interlocked Pipeline Stages) project – Stanford University.

Both projects resulted in the first VLSI implementations of RISCs, the Berkeley RISC I in 1982, and the Stanford MIPS and the Berkeley RISC II, both in 1983.

.

Features of early VLSI RISCs

Features	RISC I	RISC II	MIPS
Registers	78	138	16
Instructions	31	39	55
Addressing modes	2	2	2
Instruction formats	2	2	4
Pipeline stages	2	3	5

Early Commercial RISCs

Both RISC I/II and MIPS led to commercial RISC processors:

SUN Sparc processor derived from Berkeley RISC II processor.

MIPS Computer System Corporation established purposely to develop Stanford MIPS processor, and a series of processors appeared, including the R2000, R3000, R4000, R5000, etc.

Motorola MC88100 RISC 32-bit microprocessor, introduced in 1988 maybe first RISC produced by major CISC microprocessor manufacturer.

Later Commercial RISCs

Later RISCs also incorporated superscalar operation (executing more than one instruction in one clock cycle).

Examples

IBM RS 6000, DEC Alpha family and PowerPC family

Examples of 64-bit superscalar processors

Alpha 21164, MIPS 10 000, PowerPC 620 and the UltraSparc.

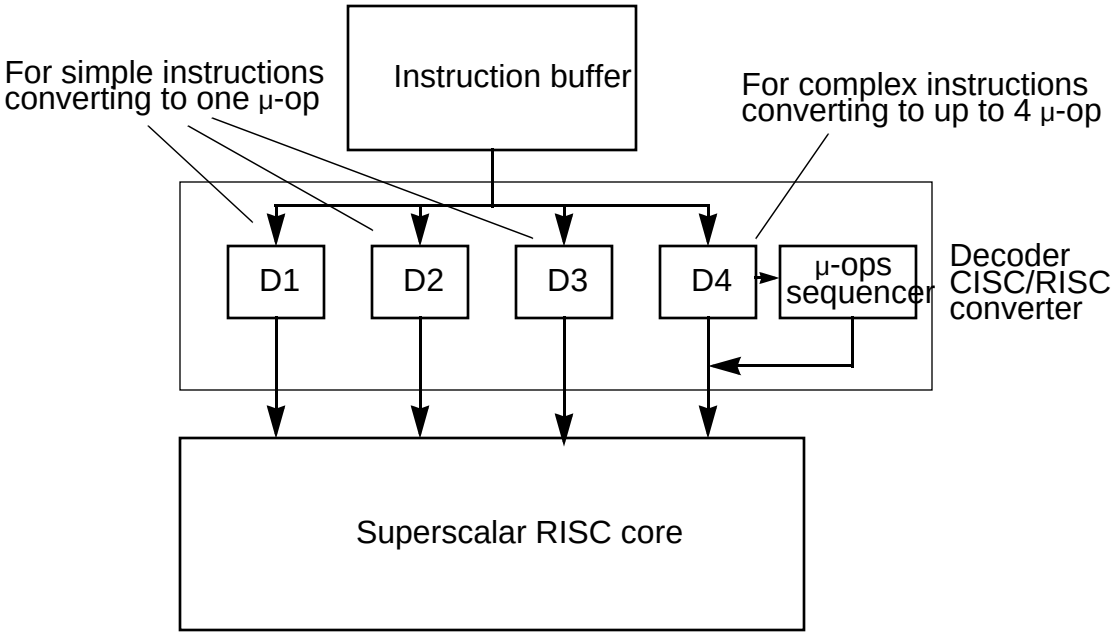
RISC-CISC

Current position (late 1990's - 2001)

The simplicity of the original RISC concept is probably been lost as most current RISCs have some complexity in their instruction set and certainly complex in their implementation to achieve their high performance (as we shall see!)

The Intel Pentium family (Peniutm Pro onwards) retained its complex instruction set externally but internally converts the complex instructions into simple RISC-like instructions.

Pentium Pro



RISC Instruction Set Design

(ITCS 3182 review)

Slide 73

Processor characteristics

Use of memory and registers

Theme is to design for maximum speed avoiding the use of memory whenever possible because memory is slower to access than registers.

L*oad* and *store* will be only instructions for accessing memory – leads to processors of this type as having a *Load/Store instruction format*.

Slide 74

For greater flexibility, three-register instruction format used for arithmetic operations.

Thirty-two integer registers commonly chosen. Compromise between providing compilers with enough registers, and having too many registers to save before context switch. Also as larger register file slower.

The thirty-two registers will be called R0–R31. Some registers given certain uses in additional to their general purpose nature.

One register will permanently store the number zero - R0. It could be any register. DEC Alpha processor, for example, uses R31 to hold zero.

Slide 75

Floating Point Instructions

All present-day processors provide for floating point numbers, and have instructions to perform arithmetic on floating point numbers.

Usually, separate registers provided for holding floating point numbers, say F0 to F31.

Always nowadays use the industry standard IEEE standard floating point formats (ANSI/IEEE 754-1985)

– either 24 bits (rarely used single precision), 32 bits (single extended precision), 64 bits (double precisions), or 80 bits (a double extended precision format). (Also 128 bits now)

One register could hold zero in floating point format permanently, say F0 (F31 in the DEC Alpha).

Slide 76

Operand and instruction size

Closely linked to allowable complexity of fabrication technology. Thirty-two bits provide reasonable precision for integers, but by late 1990s, 64-bit processors became quite common, coupled with 64-bit memory addressing.

Very few significant architectural differences between processors with different operand sizes in terms of control techniques – the main differences are in number of gates to make up the registers and ALUs etc. and number of internal data lines.

Need to specify size of the number being processed, 8 bits, 16 bits, 32 bits, 64 bits (or greater).

Size of 8 bits provided principally to handle ASCII characters.
16-bit size not particularly useful. Other sizes provided for increasing precision at expense of increasing memory requirements.

Slide 77

Instructions length of 32 bits is commonly chosen for RISCs,

Not possible to specify even a 32-bit constant or 32 bit address in a 32-bit instruction.

Slide 78

Size of the memory address

n bits in the address allows 2^n locations to be addressed.

As the allowable complexity of chips increases, so more bits are provided to address memory.

1970s – 16-bit addresses, providing for 64 Kbytes.

1980s – Increased to 20 bits (Intel 8086), 24 bits (Motorola MC68020)

1990 –1995 – 32-bit addresses providing up to 2^{32} bytes (4 gigabytes)

Thirty-two bit bits easily accomodates typical main memory sizes.

Slide 79

Present TREND

64 bits, provides for main memory up to 2^{64} bytes (18,446,744,073,709,551,616 bytes i.e. 18×10^{18} bytes). This size of main memory unlikely to be practical for many years (if ever) but ensures longevity of design.

Slide 80

Instruction formats

Register-register instructions

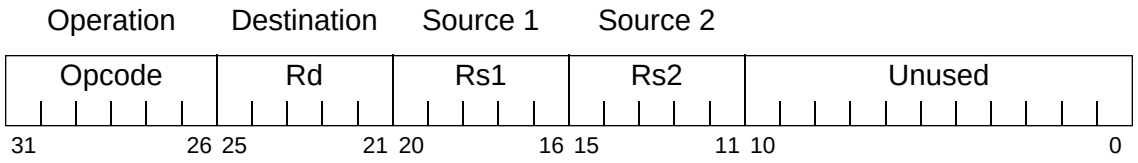
Example

```
ADD R2,R3,R4      ;R2 = R3 + R4
```

Using one register, R0, to hold zero, can create a register move operation, i.e., to copy the contents of R4 into R5:

```
ADD R5,R4,R0      ;R5 = R4 (+ 0)
```

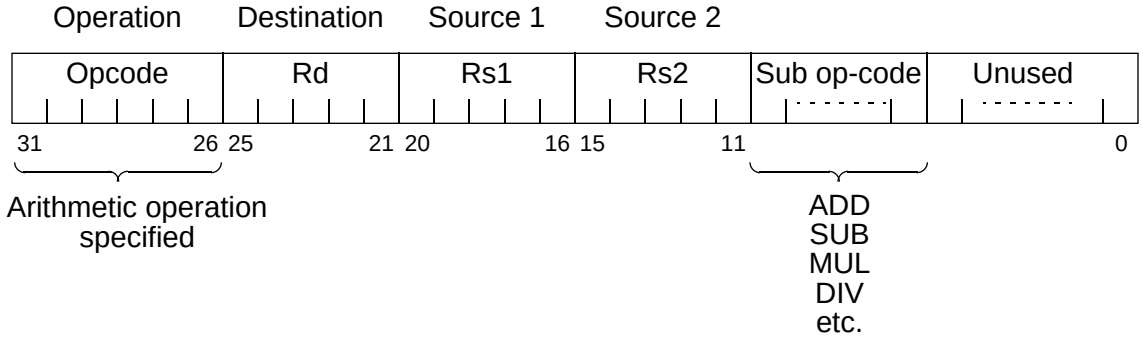
so that a specific register move instruction unnecessary.



Register-register instruction format (R-R-R format)

Sub Op-Codes

Could use remaining currently unused bits in instruction to specify a sub-operation, perhaps an operation associated with a functional unit. Then primary opcode specifies class of operations processed by functional unit.



Register-register instruction format with sub op-code

Register-register-constant format

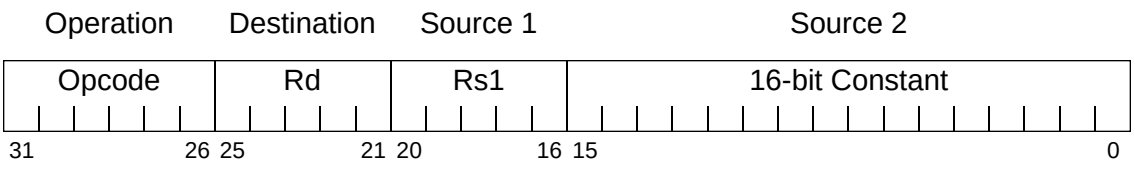
Very common requirement to be able to load or add a constant to a register.

– generally referred to *immediate addressing* because the constant is part of the instruction and immediately follows the rest of the instruction. Term *literal* is sometimes used to convey the idea of the value being literally available.

Example

```
ADD R2,R3,1234   ;R2 = R3 + 1234
```

Constant held in instruction sign-extended to no of bits of register.



Register-constant instruction format (R-R-I format)

Slide 85

Loading large constant into a register

Single instruction cannot be used to load large constant into a register.

Either:

- Use memory location with a memory load instruction, or
- Provide extra instructions to load parts of the register.

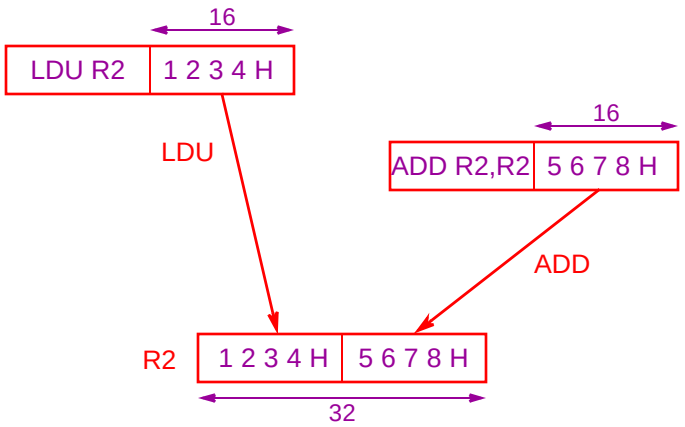
Early RISC processors with 32-bit register used extra instruction (a "load upper" instruction) for loading 32-bit constant into 32-bit register.

Slide 86

Example

To load 12345678 (hexadecimal) into a 32-bit register R2:

```
LDU R2,1234H     ;R2 = 1234H
ADD R2,R2,5678H  ;R2 = R2 + 5678H
```



This sequence not be sufficient for 64-bit registers – unfortunate connection between the instruction set and the size of registers.

Register-memory format

For loading registers from memory locations and storing registers in memory locations.

One addressing mode, register indirect addressing with offset, provides an addressing mode from which most other addressing can be created.

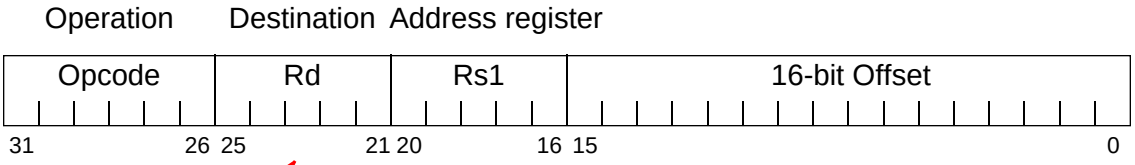
Register-memory format

Rd Rs1

Example:

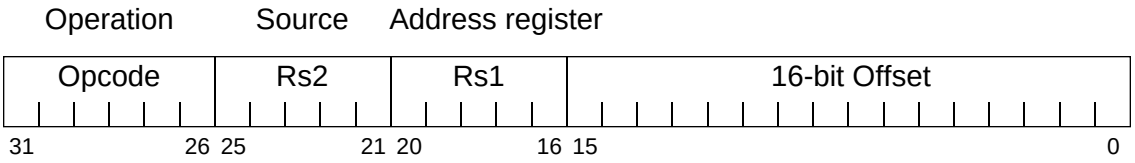
```
LD R1,100[R2]  ;Contents of memory location
               ;whose address is given by contents
      Rs1 Rs2  ;of R2 + 100 is copied in R1
ST 200[R8],R6  ;R6 is copied into memory location
               ;whose address is given by R8 + 200
```

Note: For 32-bit registers, 32 bit word transfered to/from memory (consecutive memory locations) - applies throughout.



Notice change order

(a) Load format

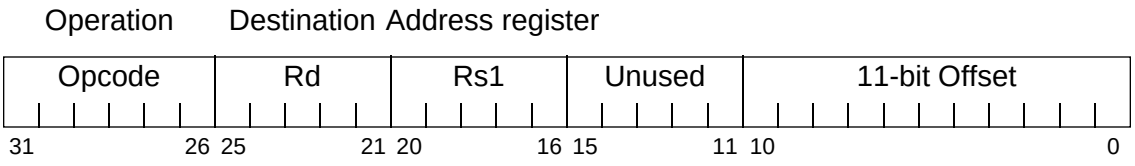


(b) Store format

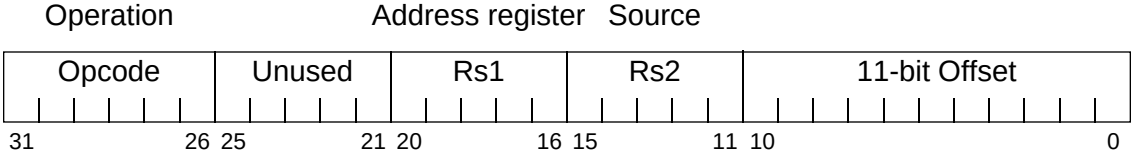
Load/store instruction formats (using R-R-I format) - version 1

Alternative Store Instruction Format

To keep address fields consistent with other instructions. Load also altered to be similar:



(a) Load format



(b) Store format

Lecture 4

Control Flow

Instructions to alter execution sequence dependent upon computed value. Needed to implement high level statements such as if, while, do-while, etc. Compilers must translate statements such as:

```
if (x != y) && (z < 0) {
    a = b + 5;
    b = b + 1;
}
```

into machine instructions. Unreasonable to have a unique machine instructions for each IF statement because of vast number of possible IF statements. Need to extract essential primitive operations for machine instructions.

Decompose into simple IF statements of the form:

```
if (x relation y) goto L1;
```

where **`relation`** is any of usual relations allowed in high level languages (<, >, >=, <=, ==, !=), i.e.:

```
if (x != y) && (z < 0) {
    a = b + 5;
    b = b + 1;
}
```

into

```
    if (x == y) goto L1;
    if (z => 0) goto L1;
    a = b + 5;
    b = b + 1;
L1:
```

More than one way of creating above IF statement.

There is more than one way of implementing:

```
if (x relation y) goto L1;
```

with machine instructions.

Here, we will start with the very common conditional code register approach and then some alternatives which may be preferable for high performance processors.

Conditional Code Register Approach

The most common is to decompose the IF statement into two machine instructions:

- one instruction that tests the boolean condition “(x relation y)” and
- a second instruction which performs the “goto L1” if the relationship is true.

The result of test of the first instruction is stored in a so-called condition code register (CCR) for the second intruction to read.

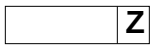
Example

The if statement:

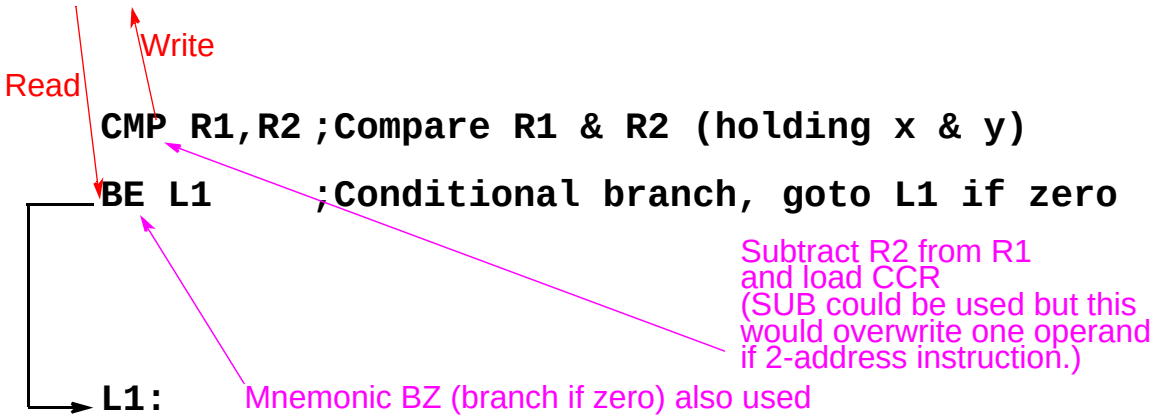
```
if (x == y) goto L1;
```

could be implemented by a sequence of two instructions:

Condition code register



Z = 1 if R1 - R2 = zero otherwise Z = 0



Conditional Code Register Flags

To cope with every possible Boolean condition, i.e. <, ≤, =, !=, > , ≥, need more than one flag in CCR, zero (Z), and negative (S for sign) necessary for basic conditions, and one conditional branch instruction for each condition.

Other flags in CCR that usually exist include carry (C), and overflow (O)

Slide 97

Conditional Branch Instructions

Mnemonic	Condition	C notation	Flags checked*
BL	Branch if less than	<	S
BG	Branch if greater than	>	$\overline{S} \cdot \overline{Z}$
BGE	Branch if greater or equal to	>=	$\overline{S}$
BLE	Branch if less or equal to	<=	$S + Z$
BE	Branch if equal	==	Z
BNE	Branch if not equal	!=	$\overline{Z}$

Logical AND

Logical OR

* assuming 2's complement representation and not taking into account any overflow conditions. Separate conditional branch instructions necessary for unsigned numbers.

Slide 98

Specifying Target Location (L1)

Mostly, condition branch instructions used to implement small changes in sequences or program loops of relatively short length.

Also good programming practice to limit sequence changes to short distance from current location to avoid difficult to read code. Helps make code *relocatable*. i.e. code can be loaded anywhere in memory without having to change the branch and other addresses.

Slide 99

PC-Relative Addressing

Number of locations from address of present (or next) instruction held in instruction as an offset.

Offset added to program counter to obtain effective address.

Slide 100

We have decomposed the IF statement:

```
if (x relation y) goto L1
```

into two sequential actions:

```
compare: (x - y);Set condition codes S,O,C,Z, etc.
branch:  if (certain condition codes set) goto L1
```

Slide 101

The problem with CCR approach

- Requires the first action (compare) to be performed completely before the second action (branch) can be started.
- The two instructions generally need to be next to each other.

Hence, limits the processor from executing instructions simultaneously or not in program order (which can improve performance).

Slide 102

PowerPC 601

Uses eight 4-bit Condition Code Registers - up to eight branch instructions each using different Condition Code Register.

Avoiding use of condition code register

Combined compare and branch instruction

An alternative which both avoids the use of a condition code register and eliminates the necessity of a sequence of two sequential instructions is to combine the two instructions into one conditional branch instruction.

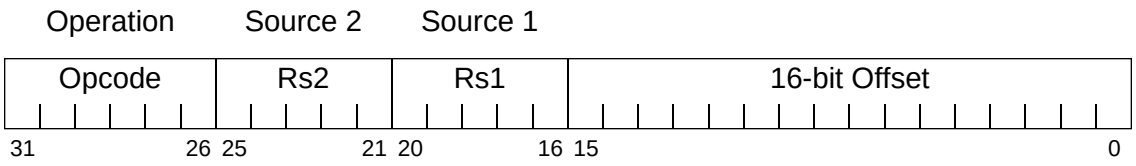
These instruction compares the contents of two registers, and branches upon a specified condition:

Combined compare and branch instructions

```
BEQ R1,R2,L1 ;Branch to L1 if R1= R2
BNE R1,R2,L1 ;Branch to L1 if R1≠ R2
BL R1,R2,L1  ;Branch to L1 if R1< R2
BLE R1,R2,L1 ;Branch to L1 if R1≤ R2
BG R1,R2,L1  ;Branch to L1 if R1> R2
BGE R1,R2,L1 ;Branch to L1 if R1≥ R2
```

A separate instruction is needed for each condition (as in the CCR approach).

Combined Compare and Branch Instruction Format



Combined test and branch instruction format (using R-R-I format)

Problems with this approach:

- Complex instruction!
- Limited space for offset to L1

Slide 107

Branch Instruction testing for zero

Testing for zero is a very common operation in programs. Could provide a "branch if zero" and "branch if not zero", instructions specifically, i.e.:

```
BEQZ  R3,L1  ;Branch to L1 if R3= 0
BNEQZ R3,L1  ;Branch to L1 if R3≠ 0
```

although it would be easy to accomplish previously with R0, i.e.:

```
BE R3,R0,L1  ;Branch to L1 if R3= 0
BNE R3,R0,L1 ;Branch to L1 if R3≠ 0
```

Advantage of having special instructions for testing for zero is there is more space in the instruction to specify L1 as a bigger offset. Also a very fast circuit could be used to test for zero.

Slide 108

In many high level statement situations, what at first sight appears to require a complex test can be reduced by a compiler to a test for zero.

For example, the C loop:

```
for(i = 0; i < 10; i++) b[i] = a[i];
```

can be written as:

```
for(i = 0; i != 10; i++) b[i] = a[i];
```

which requires a test for **(i - 10) != 0**

Code sequence could even be re-arranged to:

```
for(i = 9; i != 0; i--) b[i] = a[i];
   b[i] = a[i];
```

Using general-purpose register to hold condition codes

Instruction performs a compare operation, creating condition code values loaded into a general-purpose register specified in instruction. Subsequent branch instruction inspects register loaded with condition codes.

Allows us to separate the two instructions in the program more easily.

Example

The “set on less” instruction found on the MIPS RISC processor.

The “set on less” instruction sets the destination register to 1 if one source register is less than the other source register.

Then a “branch on not equal or not zero” can be used for the relationship “less than”, i.e.:

```
STL R3,R2,R1 ;R3 = 1 if R2 < R1
BNE R3,R0,L1 ;Branch to L1 if R3≠0, if R2 < R1
```

Jump instructions

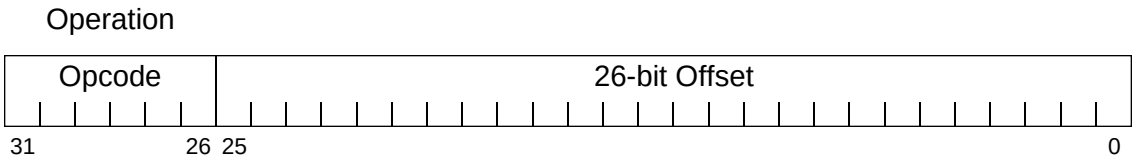
Jump instruction causes unconditional change of execution sequence to a new location. Necessary to implement more complicated IF constructs, FOR, and WHILE loops.

Using J as the opcode mnemonic, the jump instruction is:

```
J L1                ;jump to L1
```

As with branch instructions, PC-relative addressing is used.

No registers need be specified.



Jump instruction format (I format)

Jump Instruction with register-indirect addressing

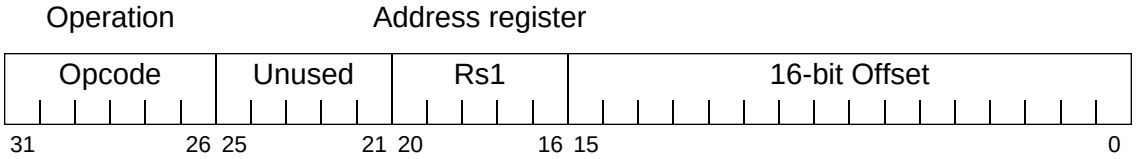
An “address register” specified in instruction holds address of location holding next instruction to be executed. Used, for example, to implement SWITCH statements in a high level language. May also be necessary for procedure return operations (see later).

```
J [R1]     ;jump to location whose address is in R1
```

or even:

```
J 100[R1]  ;jump to location whose address is in
           ;R1 plus 100.
```

Now target address specified as an absolute address, rather than as a relative address.



Jump instruction format with register indirect addressing

Procedural calls

Essential ingredient of high level language programs – the facility to execute procedures, code sequences, that are needed repeatedly through a main program, rather than duplicate the code.

Two basic issues to resolve in implementing procedures:

- A mechanism must be in place to be able to jump to procedure from various locations in *calling* program (or procedure), and to be able to return from *called* procedure to the right place in calling program (or procedure).

- A mechanism must be in place to handle passing actual parameters (arguments) to the procedure, and to return results (if a function).

Also usually when a procedure is called, registers being used by the calling procedure must be saved, so that they can be reused by the called procedure.

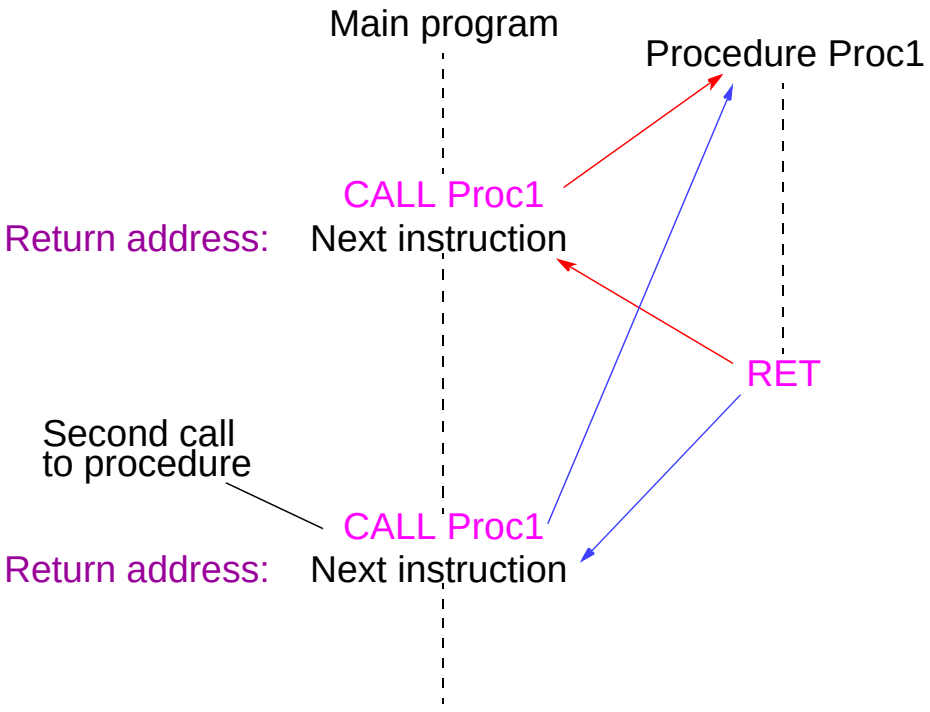
Methods to implement Procedures

CALL/RET instructions

Special machine instructions for both procedural call and procedural return in complex instruction set tradition, often called CALL and RET:.

CALL – simply an unconditional jump to the start of procedure, with the added feature that the *return address* (the address of the next instruction after the call) is retained somewhere.

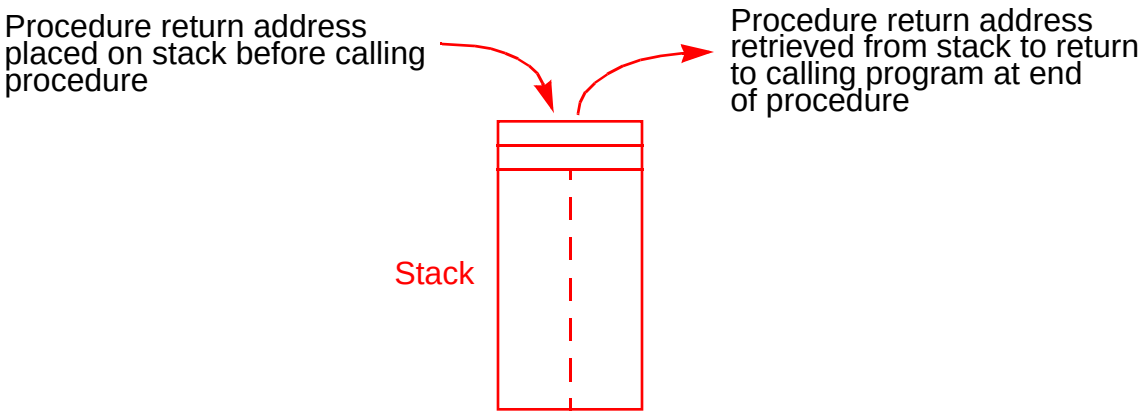
RET – to return to the main program after execution of procedure - simply an unconditional jump to the location having the address given by the return address.



Procedure calls using CALL and RET instruction

Stacks

Most common method of holding return addresses – last-in-first-out queue (LIFO), *stack*, Can be implemented in main memory or using registers within the processor. Historically, main memory stacks used because they allow almost limitless nesting and recursion of procedures.



Stack Pointer

A register called a *stack pointer* provided inside processor to hold the address of the "top" of the stack (the end of the stack where items are loaded are retrieved).

Depending upon design, stack pointer either holds address of the next free location on the stack or the address of the last item placed on the stack.

Question: Any advantage of each approach?

Examples

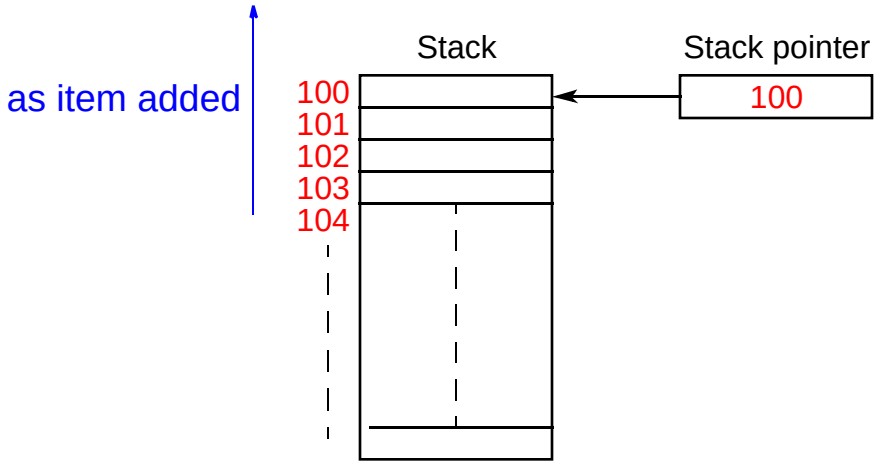
Although any register (R1 - R31) could be used in RISC, usually a convention exists:

MIPS uses R29 as a stack pointer pointing to next free location.

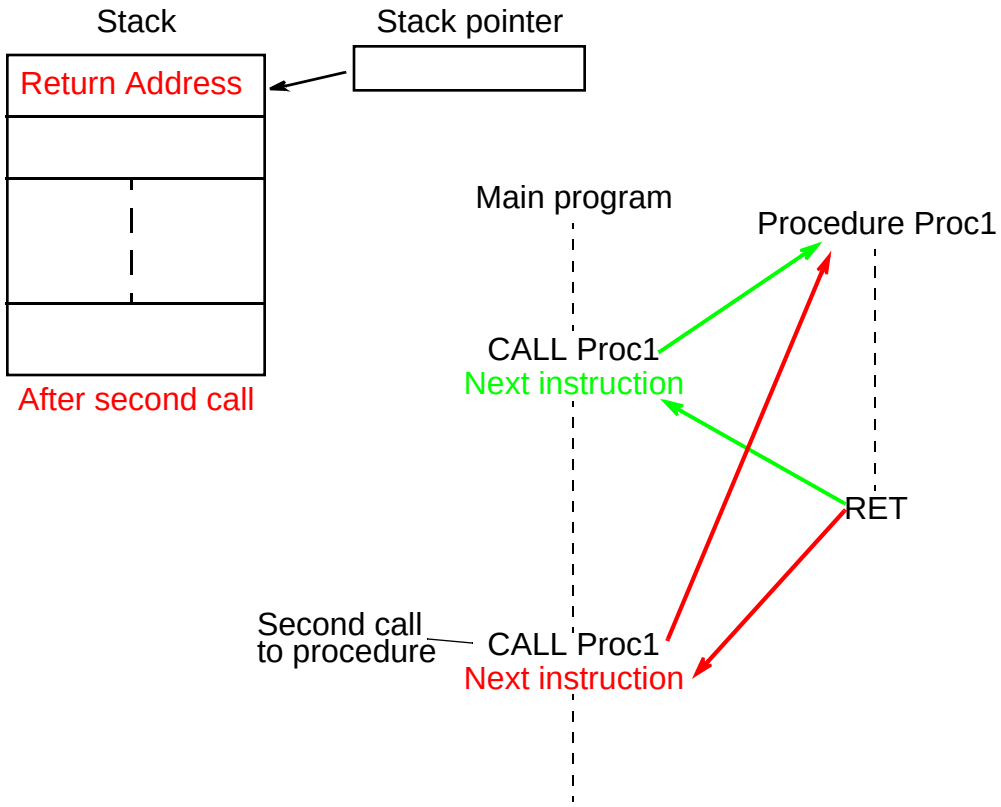
Alpha uses R30 as a stack pointer pointing to last item on stack.

Pentium has dedicated stack pointer (SP) pointing to last item on stack.

As items are placed on the stack, the stack grows, and as items are removed from the stack, the stack contracts. Although not shown in figures here, normally memory stacks are made to grow downwards, i.e., items are added to locations with decreasing addresses. Why?



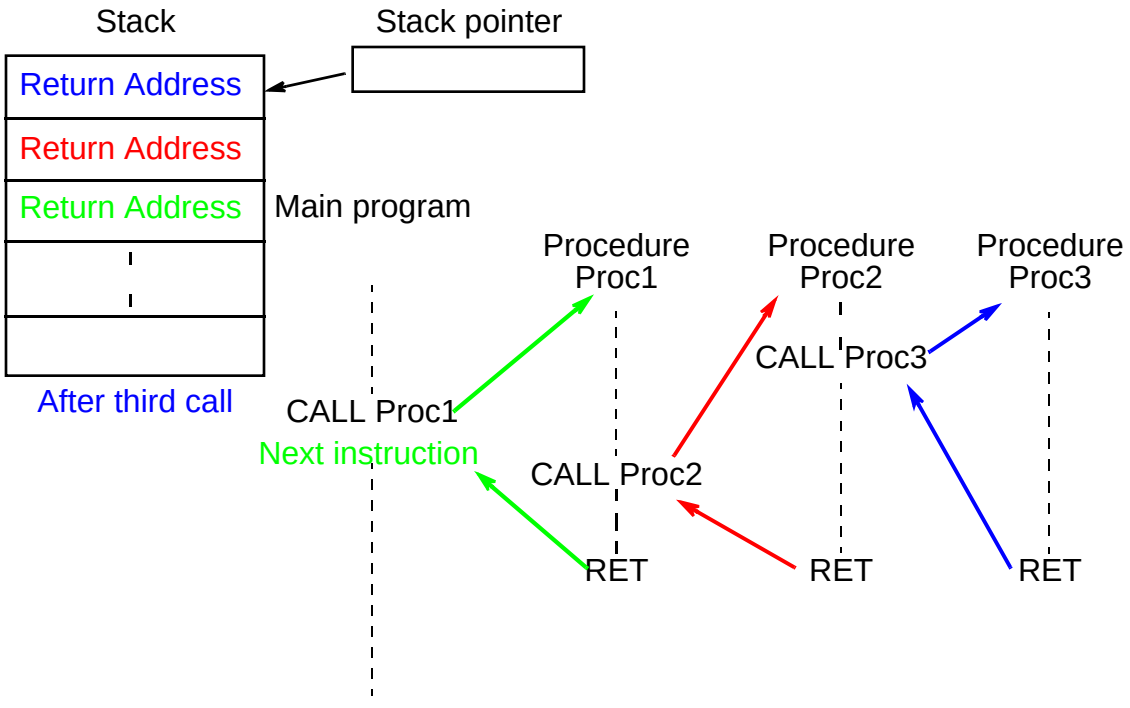
Slide 123



Slide 124

Nested Procedure Calls

Stack provides storage for each return address for nested/recursive calls.



Slide 125

Suppose 32-bit addresses stored on stack. Then, 4 bytes would be needed for each address, and stack pointer decremented by four each time an address is added to stack, and incremented by four as addresses are removed from stack.

Part of CALL instruction is to decrement the stack pointer by 4 (prior to accessing stack if stack pointer points to the last item placed on the stack).

Part of RET instruction is to increment stack pointer by 4 (after to accessing stack if stack pointer points to the last item placed on the stack).

Slide 126

Passing (actual) parameters using a stack

Stack can be used to hold actual parameters passed to called procedure and passed back from called procedure.

Processors that use CALL and RET instructions also usually provide instructions for passing items on the stack, and for taking items off the stack, called PUSH and POP instructions.

- PUSH instruction - decrements the stack pointer before (or after) an item is copied onto the stack.
- POP instruction - increments the stack pointer after (or before) an item is copied from the stack.

Before the call and return address pushed onto stack, parameters are pushed onto stack. Then, within procedure, the parameters are "popped" off the stack.

Saving registers

Stack can be used to save the contents of registers prior to the call (or immediately after the call inside the procedure).

Upon return (or immediately before) the registers can be restored from the contents of the stack.

Saving/restoring registers probably best done inside called procedure.

Stack Frame

Convenient to specify the group of locations on stack which contains all the parameters, results, and return address relating to one procedure as a *stack frame* and have another pointer (*frame pointer*) to point to the current stack frame.

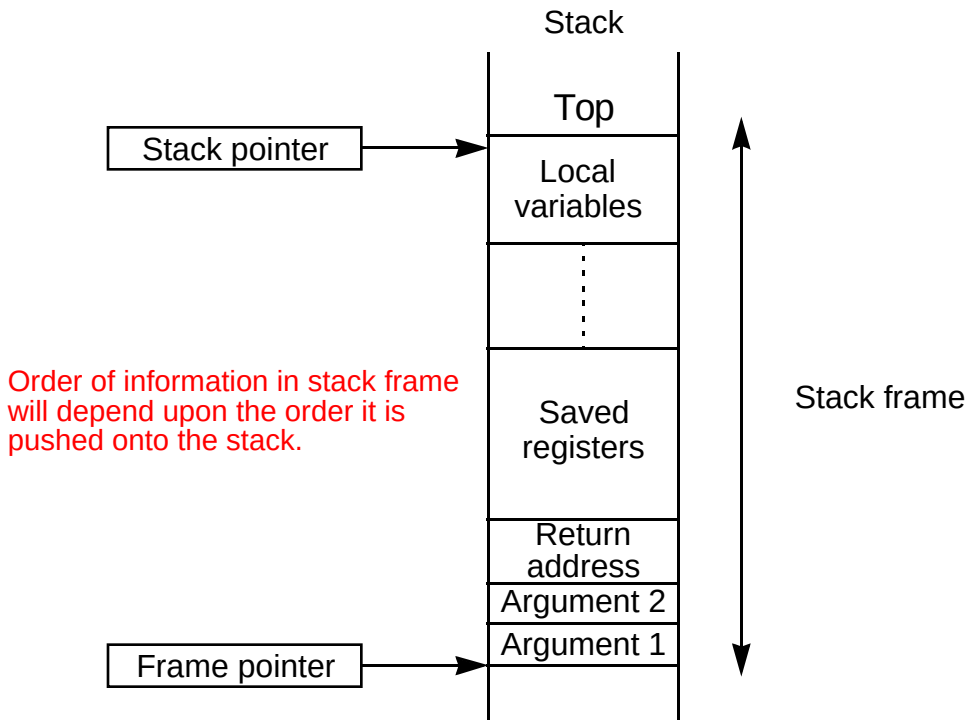
Examples

MIPS processor uses R30 as a stack frame pointer

Alpha processor uses R29 as stack frame pointer

Intel Pentium has a so-called base pointer (BP) that pointers to stack

Stack Frame



Order of information in stack frame will depend upon the order it is pushed onto the stack.

Co-routines, those routines that call each other alternately, also do not need to use a stack for the return address. A single register can be used to hold the return address. The DEC Alpha processor has a specific instruction for handling co-routines.

Problem: Work out how the registers can be used to hold the return addresses.

Slide 131

Allocated registers – Some processor designs allocate certain registers for local use within a procedure and others as global registers available throughout the program.

Compilers and programmers are intended to comply with these allocations.

Slide 132

Using Registers

Results could be returned faster using registers.

A stack is really only needed for nested or recursive procedures.

The stack frame of a procedure that does not call another procedure (or itself) could easily be held in processor registers.

Jump and Link Instruction

"jump and link", JAL, will jump to the location whose address is specified in the instruction and will store the return address in R31.

Return - simply unconditional jump to location whose address given in R31.

For nesting, R31 will be stored in a memory stack, using another register as a stack pointer, R29.

Jump and Link Code

The code required would look like:

```
SUB R29,R29,4  ;Decrement stack pointer (4 bytes)
ST  [R29],R31  ;Store last return address on stack
JAL Proc_label;jump to proc_label, and store
               ;return address in R31
LD R31,[R29]   ;After return from call, restore
               ;previous return address in R31
ADD R29,R29,4  ;Increment stack pointer
```

Slide 135

To return at the end of the procedure, we simply have:

```
J [R31]     ;jump to location whose address is in
            ;R31
```

Slide 136

To store parameters as well on stack before the call, we would have:

```
SUB R29,R29,4
ST  [R29],R31
SUB R29,R29,4
ST  [R29],parameter1
SUB R29,R29,4
ST  [R29],parameter2
JAL Proc_label
```

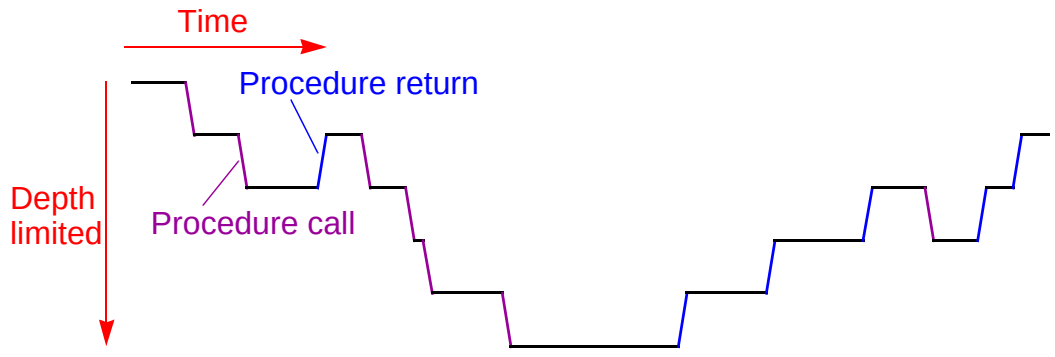
Register Window

Berkeley RISC project introduced concept of providing internal registers called the *register window* to simplify and increase speed of passing parameters and to provide local registers for each procedure.

Idea adopted in SUN Sparc processor (which followed Berkeley design).

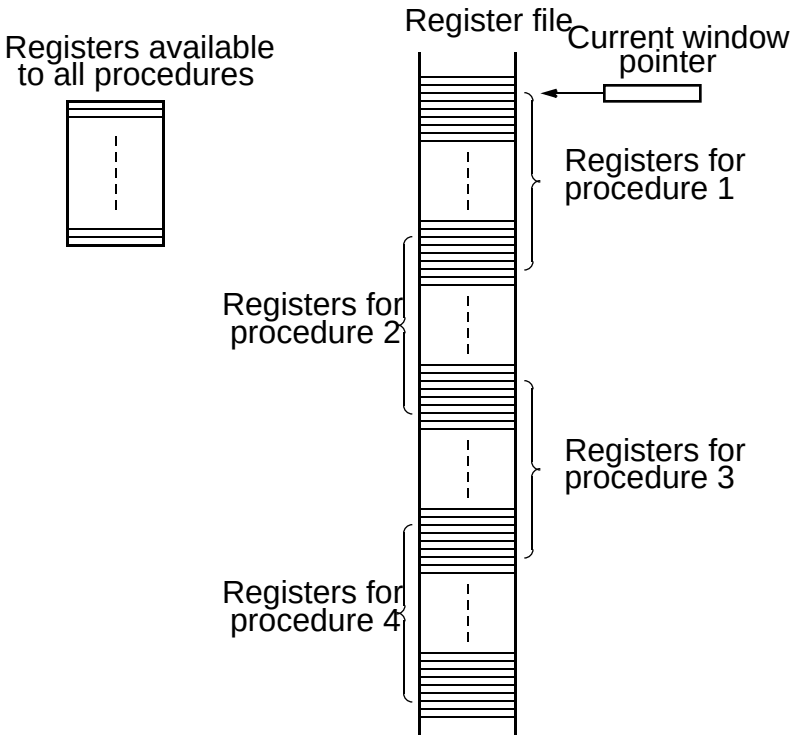
Characteristics of Procedure Calls

A procedure may call another procedure and that procedure may call another one, but the depth of procedure calls is usually limited over period of time to perhaps 10 or 11 levels maximum.



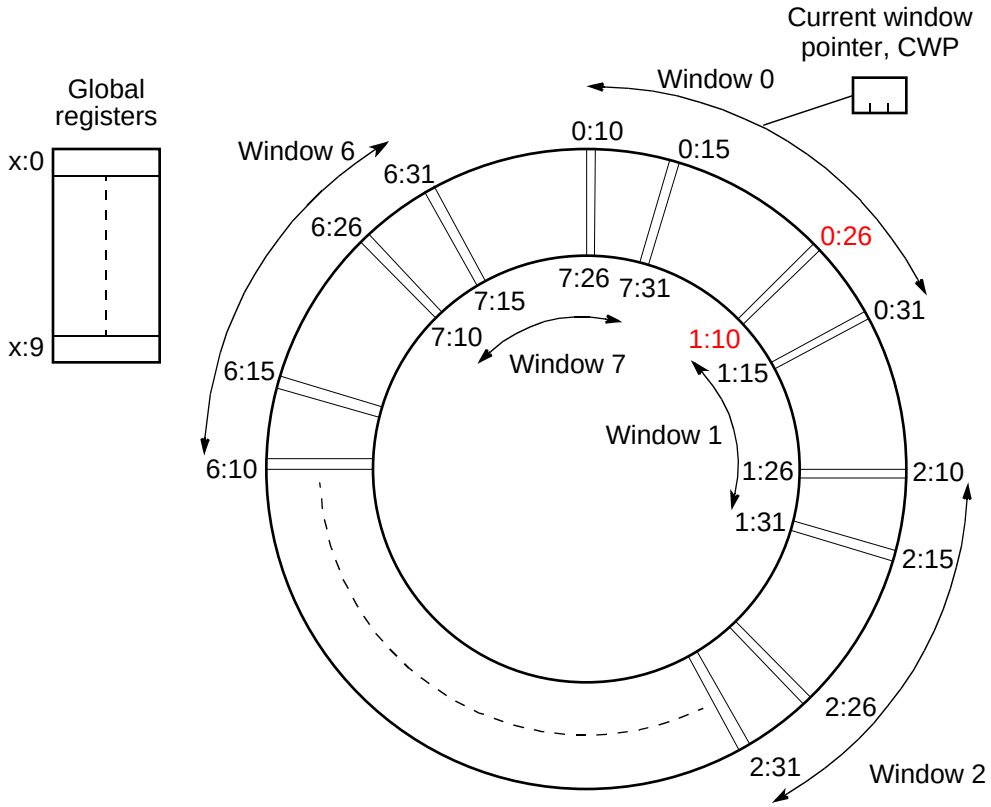
The register window mechanism takes advantage of this.

Slide 139



Register window

Slide 140



RISC II register window

Slide 141

SUN Sparc Processor

R0 to R7	8 global registers, available from all procedures (called g0 to g7 in assembly language)
R8 to R15	8 registers available to receive parameters and return results (called i0 to i7 in assembly language)
R16 to R23	8 registers available for local variables (called l0 to l7 in assembly language)
R24 to R31	8 registers available to pass parameters to next procedure and for returning values (called o0 to o7 in assembly language)

Instructions provided to adjust current window pointer:

save	Advances current window pointer (can also increment stack pointer to create space for local variables)
restore	Move current window pointer backwards

Also jump and link and ret instructions

Slide 142

Summary of register window mechanism

Advantages

- No major change necessary to instruction set -- 32 registers visible at any instant
- Fast -- uses internal registers rather than a stack

Disadvantages

- Need to handle overflow conditions -- still need a stack for this
- Fixed number of registers available for passing parameters and local variables
- Could be that existing internal first level data cache (see later) achieves similar performance making the mechanism now unnecessary.

Exercise

The gcc SUN compiler will produce assembly language output by including the **-S** flag, i.e.

gcc -S prog1.c

produces the assembly language file prog1.s. Compile the C program:

```
#include <stdio.h>
void main {
int i, sum;
  sum = 0;
  for (i = 0; i < 10; i++) sum = sum + i;
     printf("%d\n",sum);
}
```

Explain the assembly language code. Compile with the **-O** optimization flag and without it. (You will find without optimization, the register window is not used!)

Lecture 6

Pipelined Processor Design

Slide 145

Pipelining

Basic techniques to improve the performance - always applied in high performance systems. Adapted in RISCs and all processors.

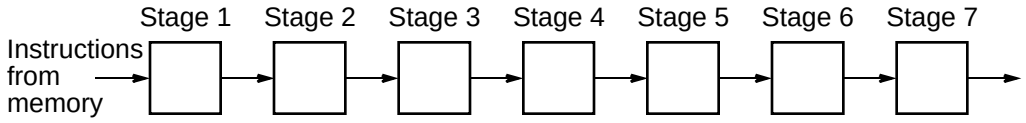
Operation of the processor divided into number of steps, e.g.:

1. Fetch instruction.
2. Fetch operands.
3. Execute operation.
4. Store results

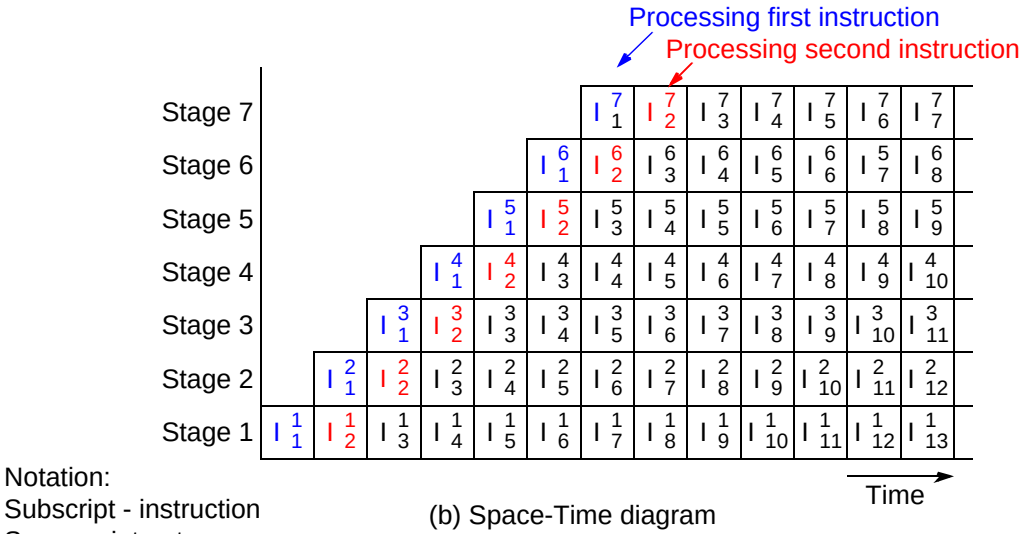
or more steps. Each step is performed by a separate unit (stage).

Slide 146

Processor Pipeline Space-Time Diagram



(a) Stages



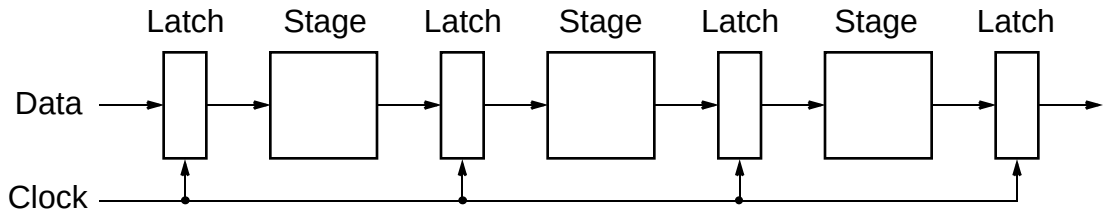
Notation:
Subscript - instruction
Superscript - stage

(b) Space-Time diagram

Slide 147

Pipeline Data Transfer

Usually, pipelines are designed using latches between stages to hold the information being transferred from one stage to the next. This transfer occurs in synchronism with a clock signal:

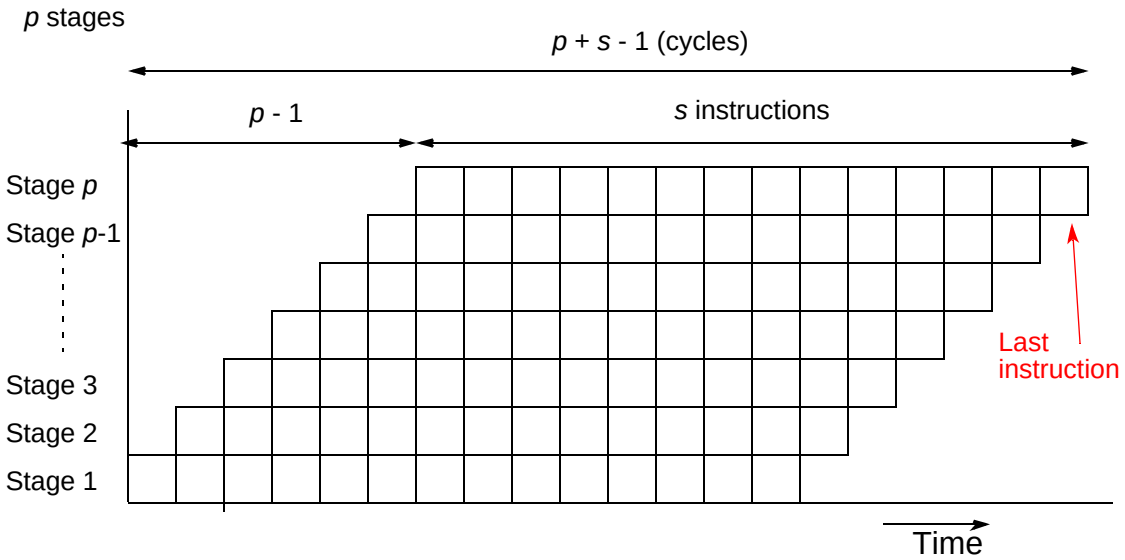


Slide 148

Processing time

Time to process *s* instructions using a pipeline with *p* stages

= *p* + *s* - 1 cycles



Speed-Up

How much faster using pipeline rather than a single homogeneous unit?

Speed-up available in a pipeline can be given by:

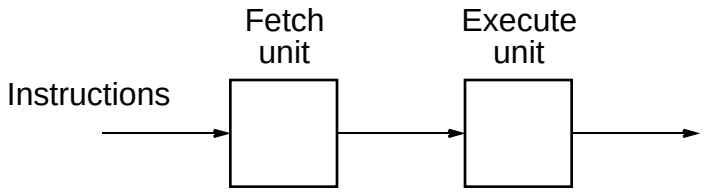
$$\text{Speed-up} = \frac{T_1}{T_p} = \frac{sp}{p + s - 1}$$

Note: This does not take into account extra time due to latches in pipeline version.

Potential maximum speed-up is p, though only be achieved for an infinite stream of tasks ($s \rightarrow \infty$) and no hold-ups in the pipeline.

An alternative to pipelining - using multiple units each doing the complete task. Units could be designed to operate faster than the pipelined version, but the system would cost much more.

Two Stage Fetch /Execute Pipeline



(a) Fetch/execute stages

EX		Execute 1st instruction	Execute 2nd instruction	Execute 3rd instruction
IF	Fetch 1st instruction	Fetch 2nd instruction	Fetch 3rd instruction	Fetch 4th instruction

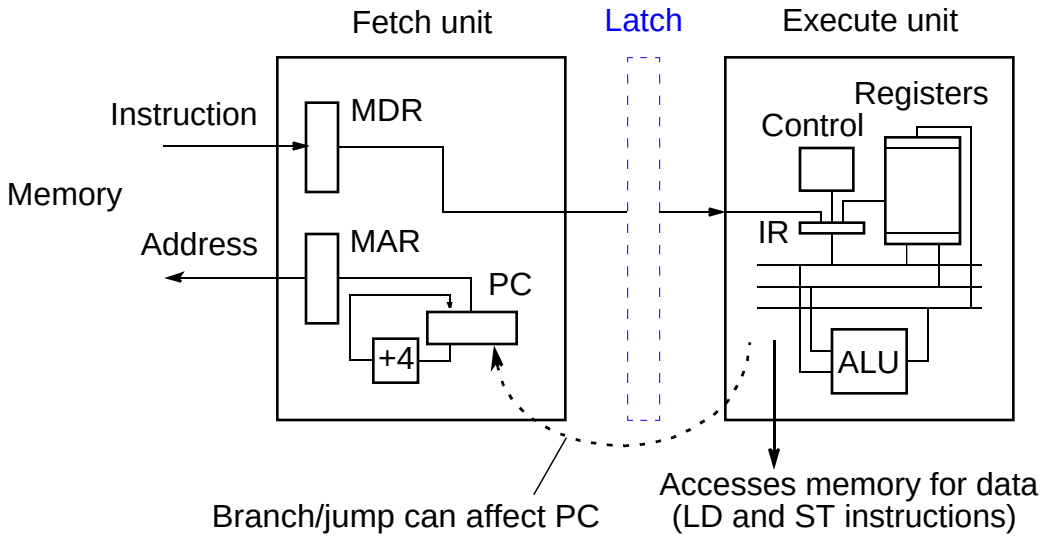
Time

IF = Fetch unit
EX = Execute unit

(b) Space-time diagram with ideal overlap

Slide 151

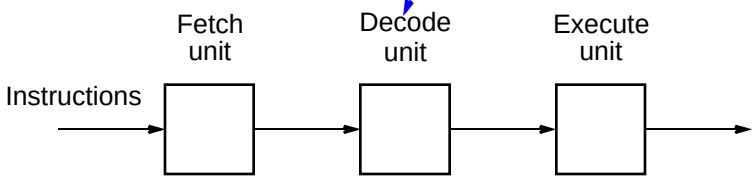
A Two-Stage Pipeline Design



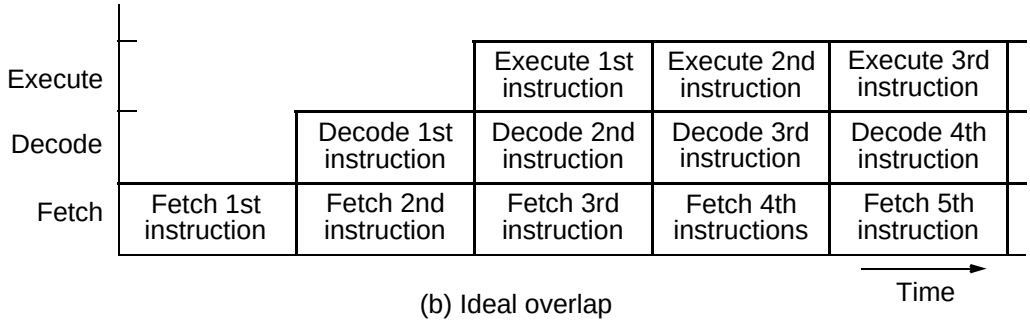
Slide 152

Fetch/decode/execute pipeline

Recognizes instruction - separates operation and operand addresses
Relevant for complex instruction formats

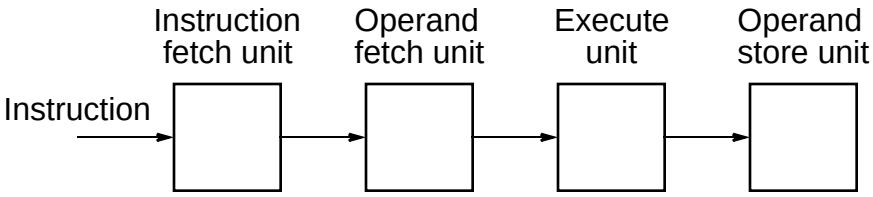


(a) Fetch/decode/execute stages

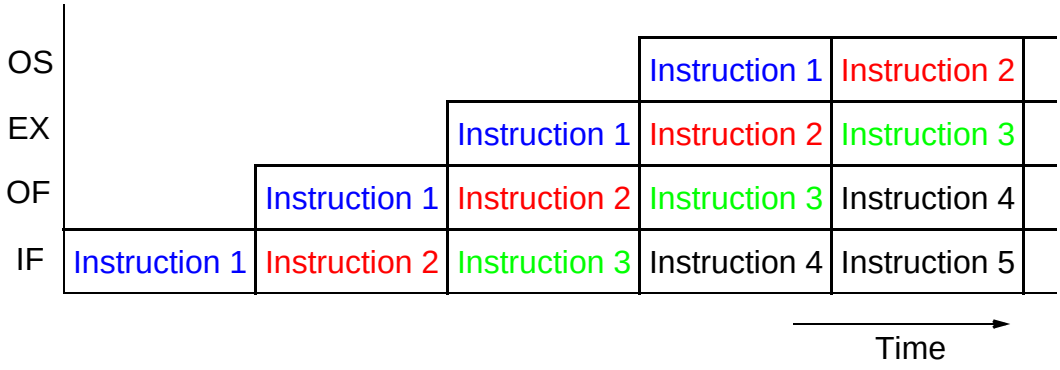


(b) Ideal overlap

Four-Stage Pipeline



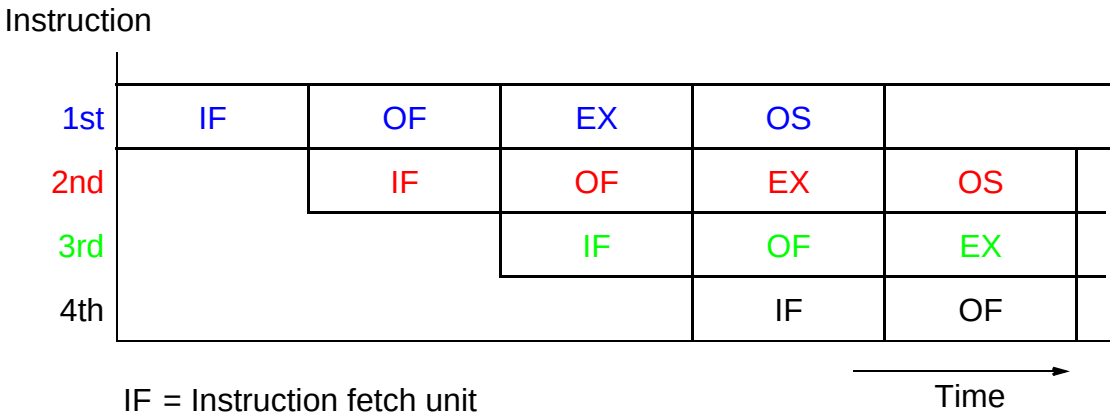
Four-Stage Pipeline Space-Time Diagram



IF = Instruction fetch unit
OF= Operand fetch unit
EX= Execute unit
OS= Operand store unit

Four-stage Pipeline "Instruction-Time Diagram"

An alternative diagram:



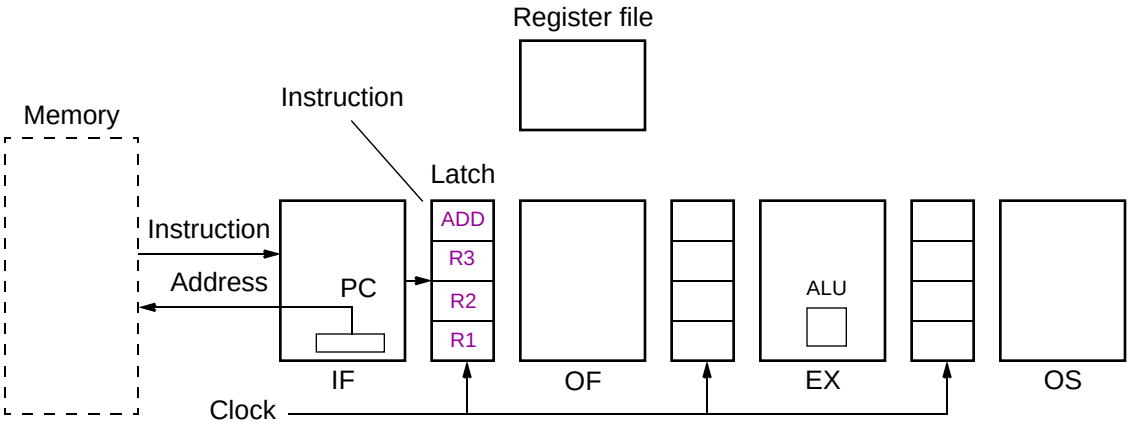
IF = Instruction fetch unit
OF= Operand fetch unit
EX = Execute unit
OS = Operand store unit

Information Transfer in Four-Stage Pipeline

Register-Register Instructions

ADD R3, R2, R1

After instruction fetched:

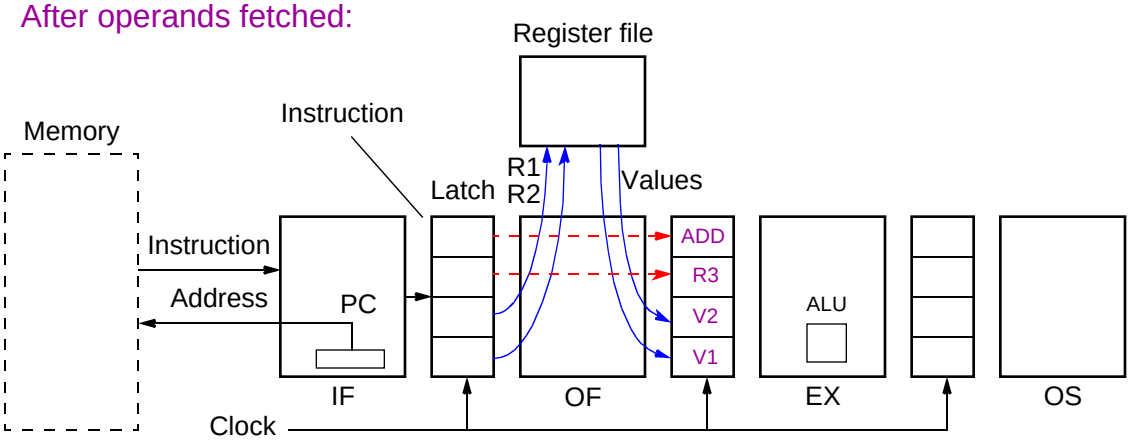


Information Transfer in Four-Stage Pipeline

Register-Register Instructions

ADD R3, R2, R1

After operands fetched:

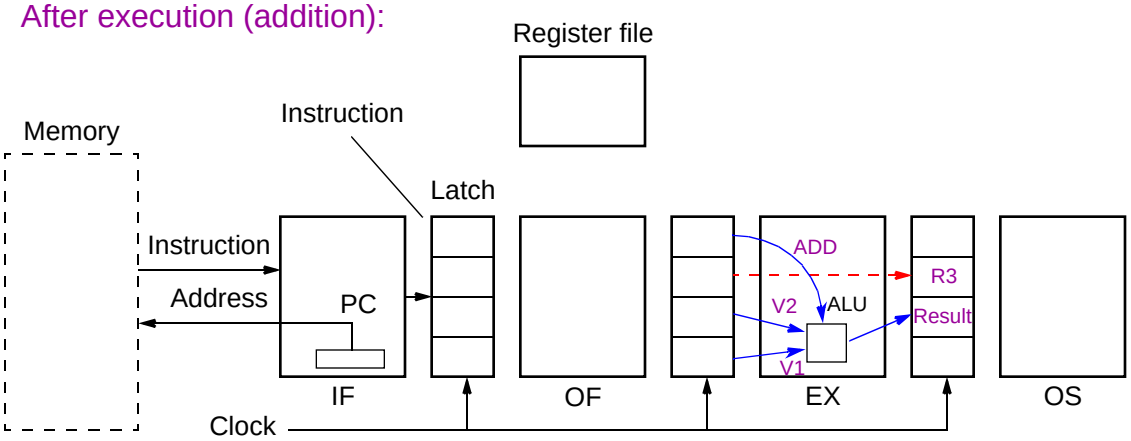


Information Transfer in Four-Stage Pipeline

Register-Register Instructions

ADD R3, R2, R1

After execution (addition):

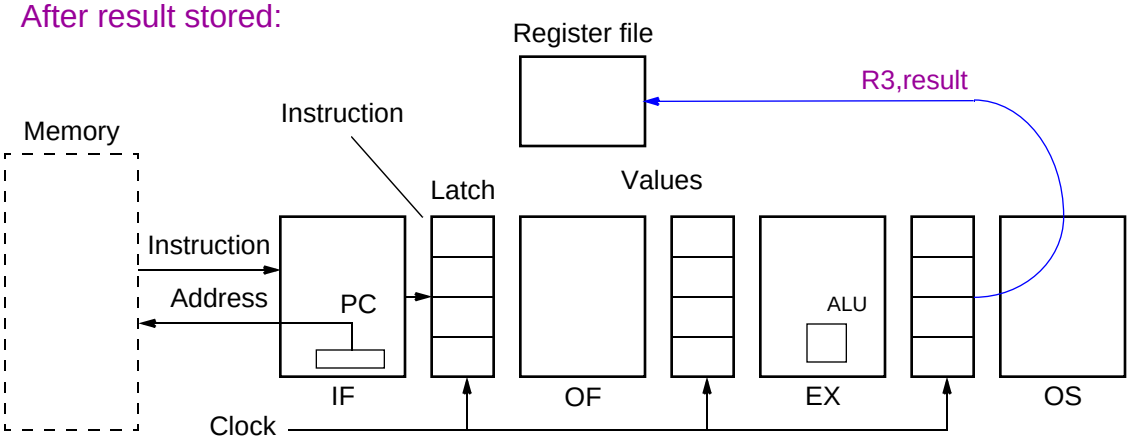


Information Transfer in Four-Stage Pipeline

Register-Register Instructions

ADD R3, R2, R1

After result stored:

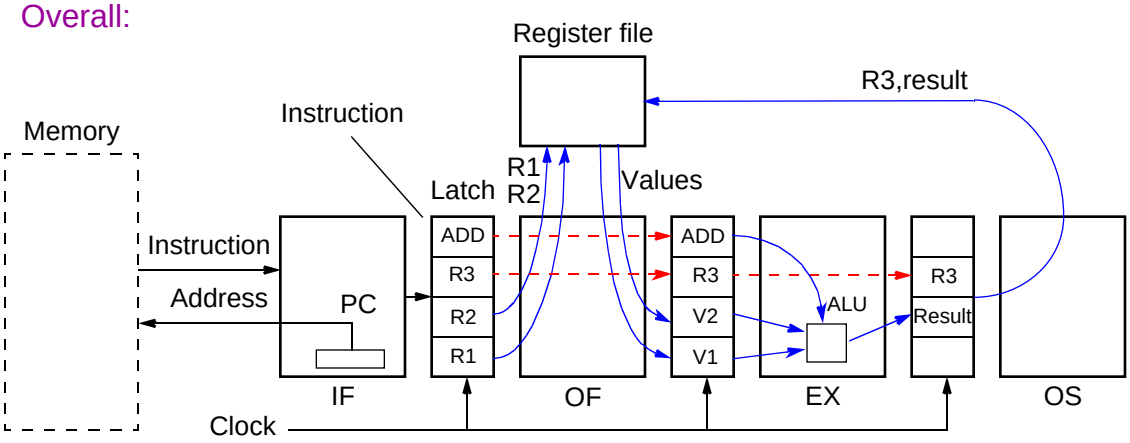


Information Transfer in Four-Stage Pipeline

Register-Register Instructions

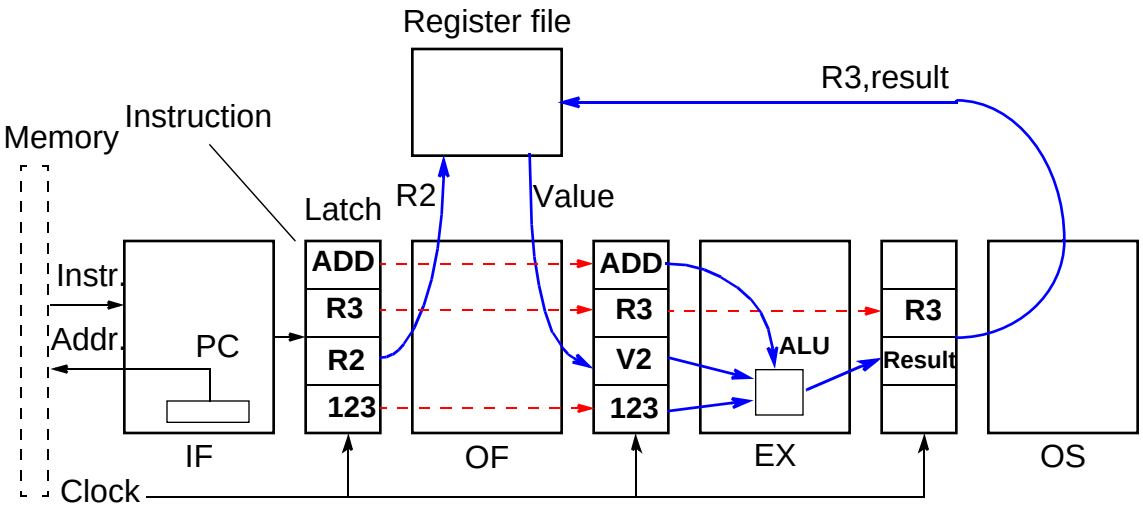
ADD R3, R2, R1

Overall:

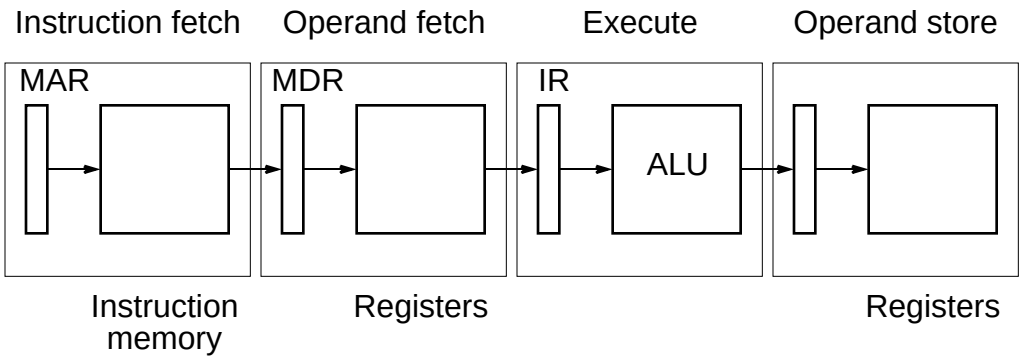


Register-Constant Instructions (Immediate addressing)

ADD R3, R2, 123



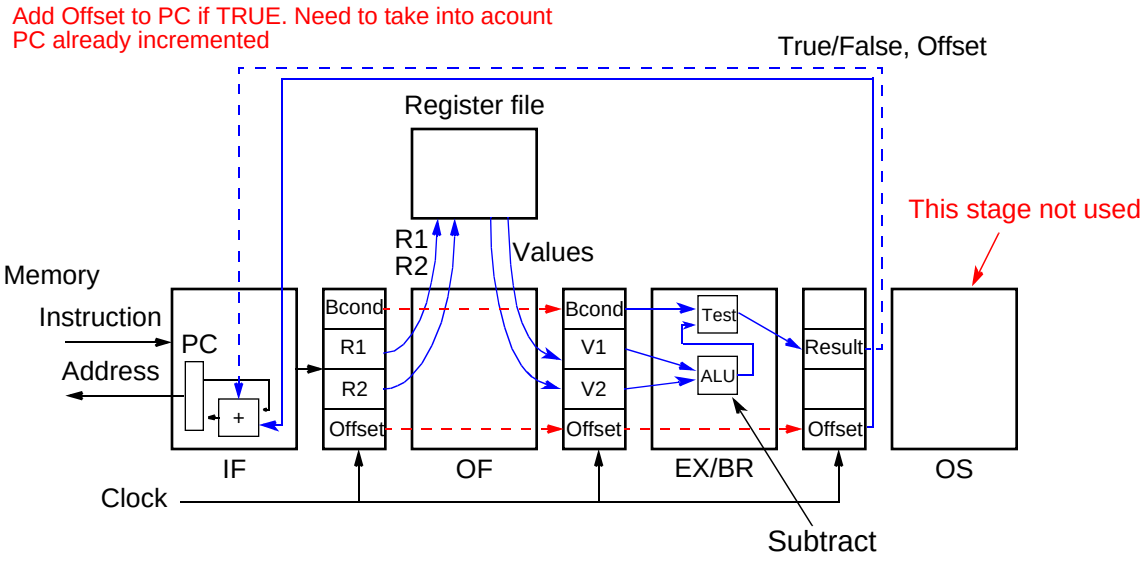
Alternative way of depicting pipeline showing data register twice



Four-Stage Pipeline

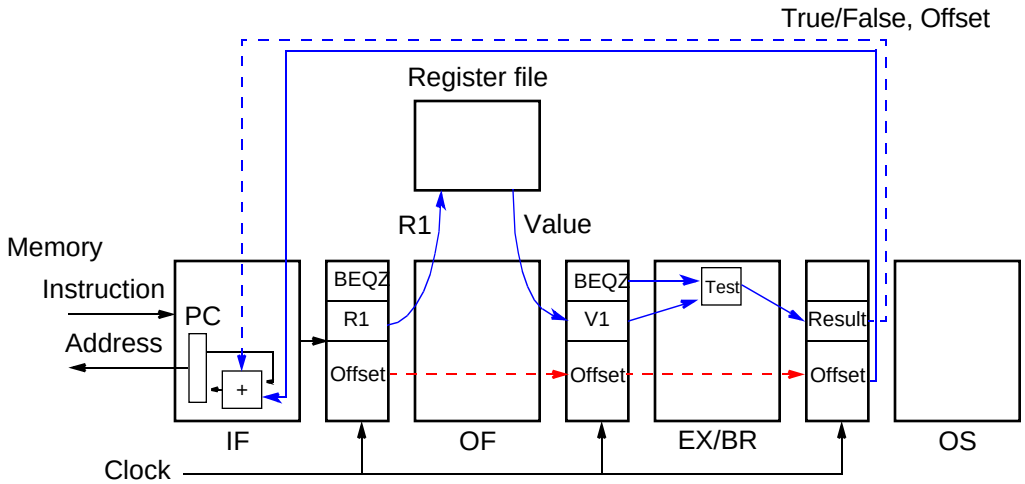
Branch Instructions

Bcond R1, R2, L1



Simpler Branch Instruction

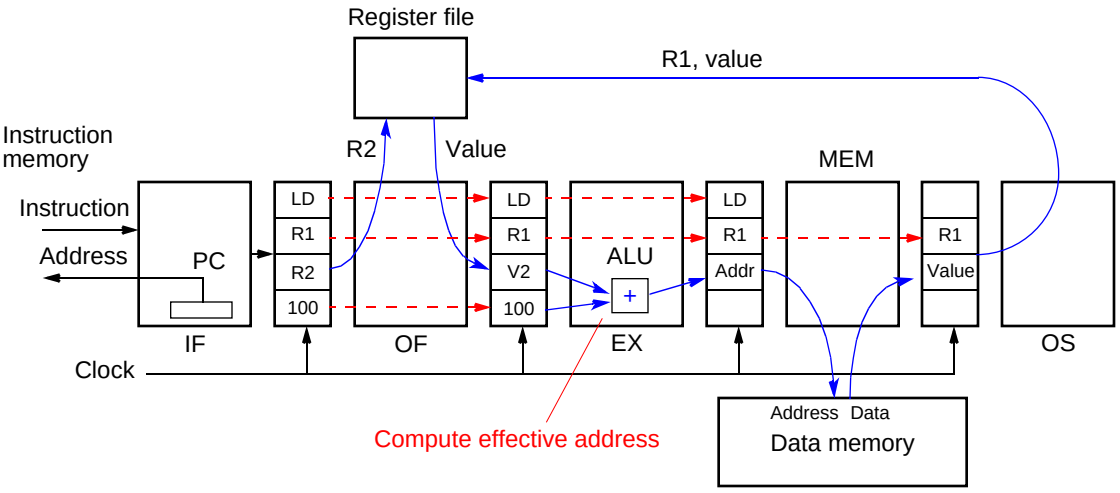
BEQZ R1, L1



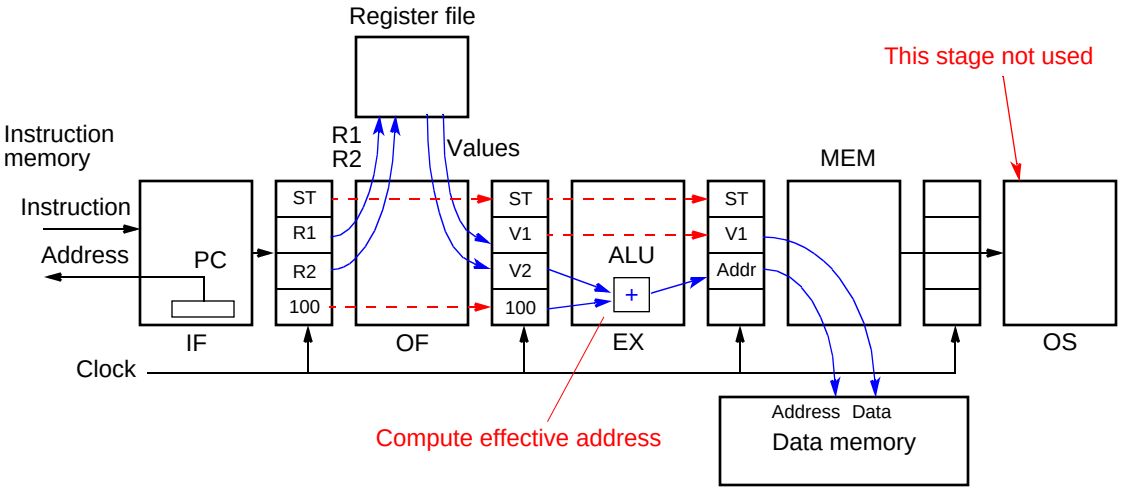
Load and Store Instructions

Need at least one extra stage to handle memory accesses. Early RISC processor arrangement was to place memory stage (MEM) between EX and OS as below. Now a five-stage pipeline.

LD R1, 100[R2]

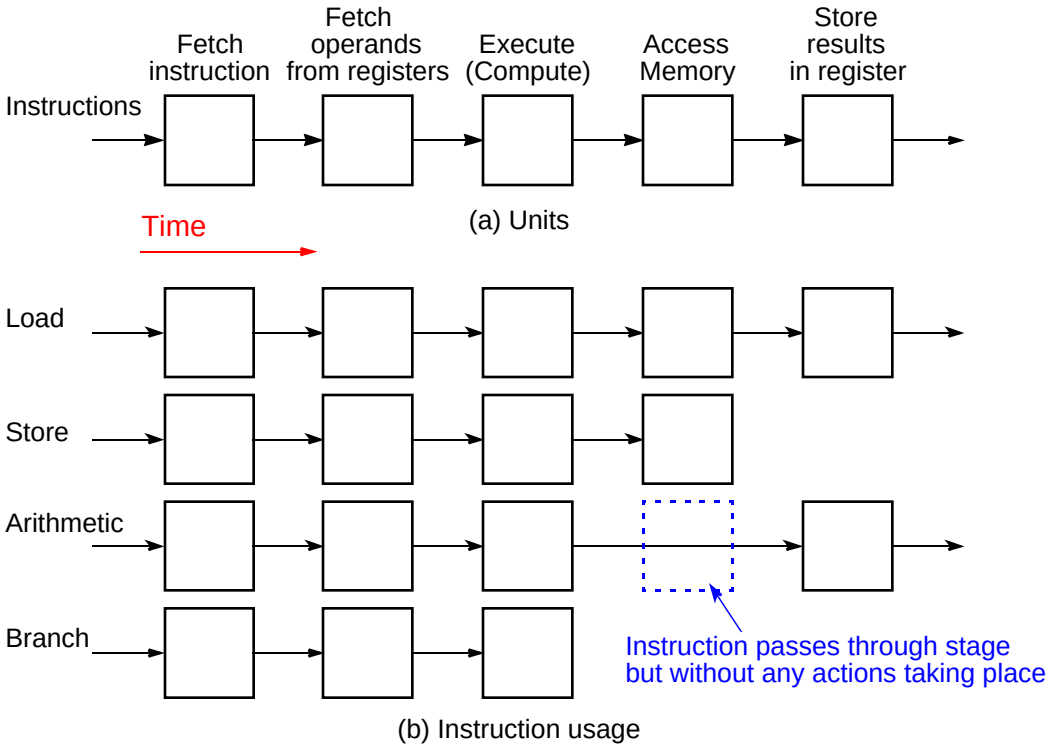


ST 100[R2], R1



Note: Convenient to have separate instruction and data memories connecting to processor pipeline - usually separate *cache* memories, see later.

Usage of Stages



Number of Pipeline Stages

As the number of stages is increased, one would expect the time for each stage to decrease, i.e. the clock period to decrease and the speed to increase.

However one must take into account the pipeline latch delay.

Optimum Number of Pipeline Stages*

Suppose one homogeneous unit doing everything takes T time units.

With p pipeline stages with equally distributed work, each stage takes T/p.

Let t_L = time for latch to operate.

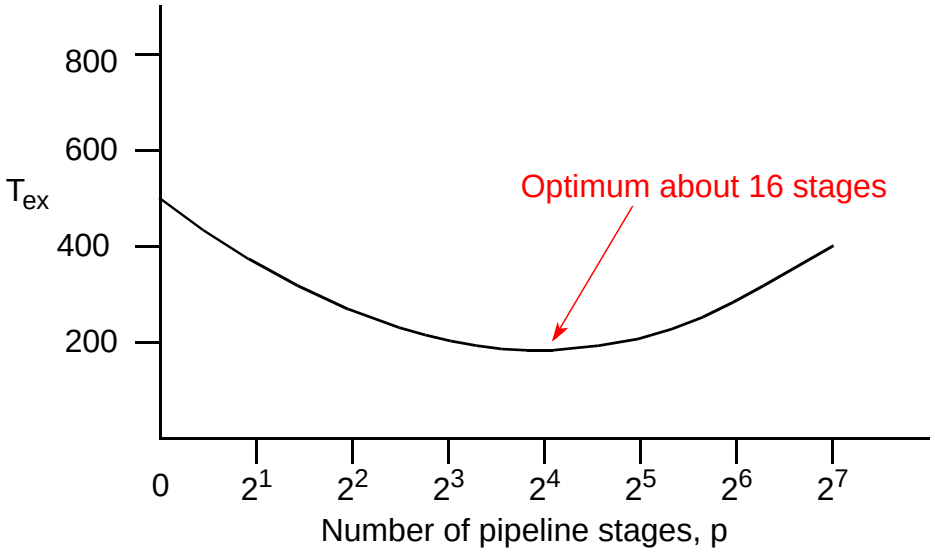
Then:

$$\text{Execution time } t_{ex} = (p + s - 1) \times (T/p + t_L)$$

* Adapted from "Computer Architecture and Implementation" by H. G. Cragon, Cambridge University Press, 2000.

Optimum Number of Pipeline Stages*

Typical results (T = 128, t_L = 2):



To get optimal value:

Differentiate t_{ex} with respect to p, set to zero and solve. Get:

$$p_{optimum} = \sqrt{\frac{T(s-1)}{t_L}}$$

* Adapted from "Computer Architecture and Implementation" by H. G. Cragon, Cambridge University Press, 2000.

5-stage pipeline represents an early RISC design - "underpipelined"

Most recent processors have more stages.

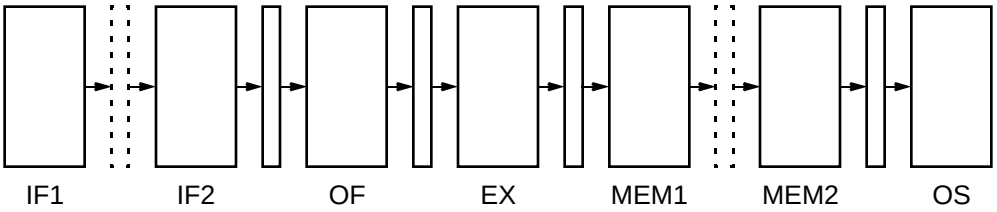
Examples

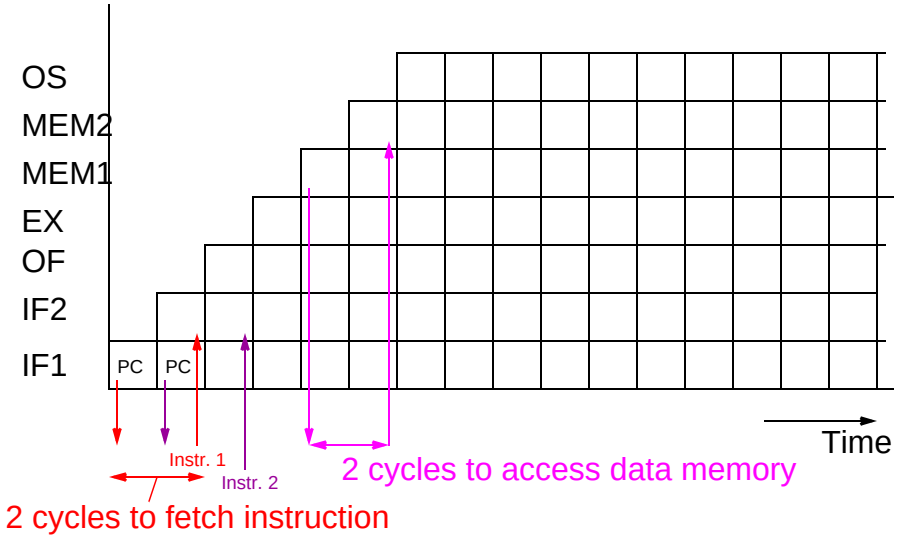
Assignment: Find out for me!

Dealing with memory access taking more time than other operations

Example

Seven-stage instruction pipeline with two stages for accessing memory:





Need memory capable of above operation (i.e. interleaved memory)

Lecture 7

Instruction Pipeline Hazards

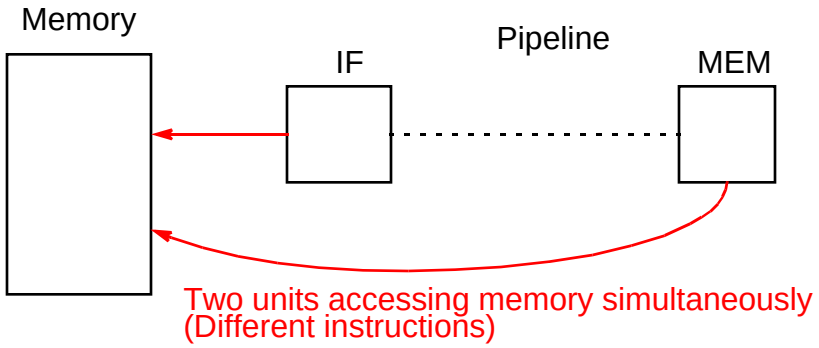
Major causes for breakdown or hesitation of an instruction pipeline:

1. Resource conflicts.
2. Control dependencies
3. Data dependencies between instructions

Resource conflicts

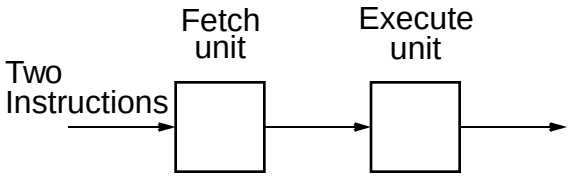
Occur when a resource such as memory, registers, or a functional unit is required by more than one instruction at the same time.

Example - memory conflict

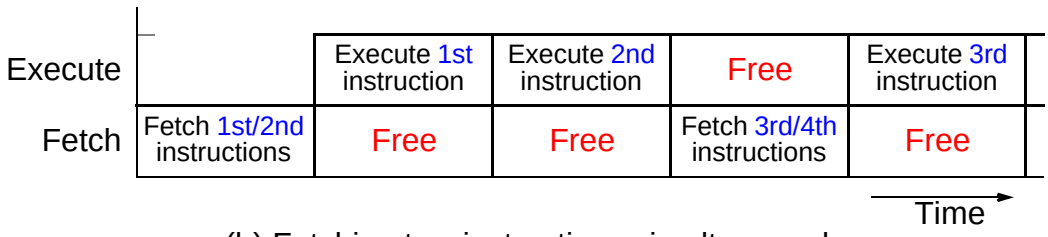


Early solution

Fetch two instructions and then execute them sequentially.



(a) Fetch/execute stages



(b) Fetching two instructions simultaneously

Fetch/execute overlap

Lower efficiency
CPI = 3/2

Slide 179

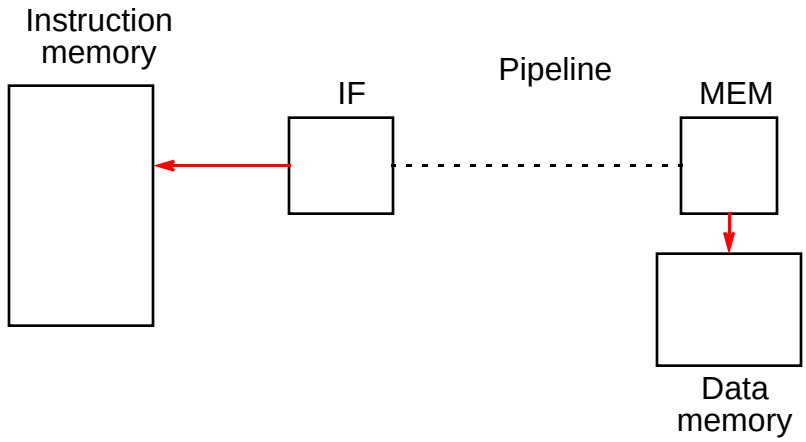
Better Solutions

Resource conflicts can always be resolved by duplicating the resource, for example having two memory module ports or two functional units.

To eliminate main memory/cache memory conflicts

- Have only load and store instructions accessing memory - then a single pipeline unit to access data memory ok, and
- Separate pipeline units for reading/writing data and for instruction fetch.

Slide 180



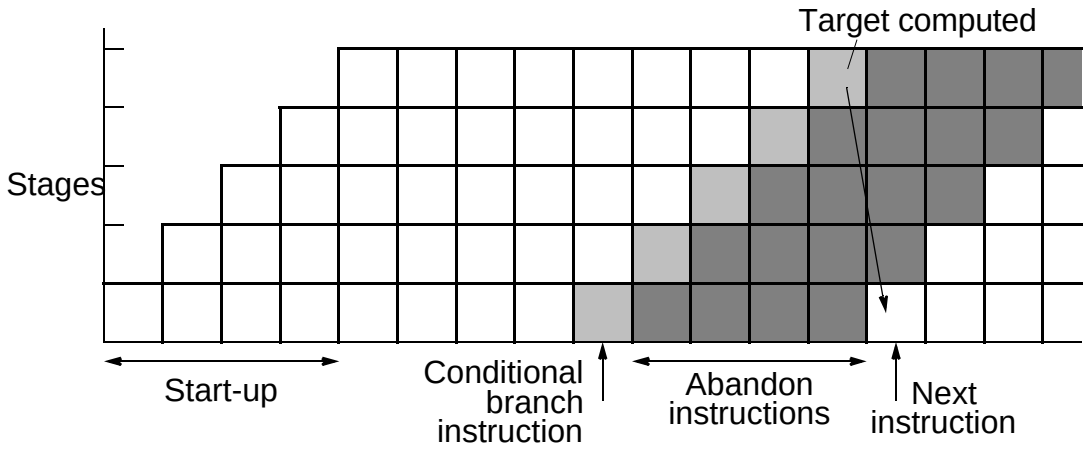
Control Dependencies

Instruction execution is dependent upon a condition existing in the program:

```
if (condition) {
statements;
}
```

i.e by the use of conditional branch instructions.

Branch instructions in a pipeline



Effect of conditional branch instruction in pipeline when instruction causes change of sequence - have to abandon already fetched instructions.

Slide 183

Typically, 10–20 per cent of instructions in a program are branch instructions.

Example of effect

- Five-stage pipeline
- 10 ns pipeline clock period
- Instruction which subsequently cleared the pipeline at the end of its execution occurred every ten instructions

For a continuous stream of instructions, average instruction processing of ten instructions:

$$\frac{9 \times 10 \text{ ns} + 1 \times 50 \text{ ns}}{10} = 14 \text{ ns}$$

i.e. a 40 per cent increase in instruction processing time.

Slide 184

Conditional branch instructions used in programs for:

1. Creating repetitive loops of instructions, terminating the loop when a specific condition occurs (loop counter = 0 or arithmetic computational result occurs).

2. To exit a loop if an error condition or other exceptional condition occurs.

Branch usually occurs in 1 when the terminating test is done at the end of the loop (as in DO–WHILE or REPEAT–UNTIL statements)

Does not usually occur in 2 or when the terminating test is done at the beginning of a loop (as in FOR and WHILE statements).

Slide 185

To implement the FOR loop **FOR (i=0;i<=100;i++) loop body** we might have:

```
     SUB R4,R4,R4
     ADD R5,R0,100
L2:  CMP R4,R5
     BG  L1          ;Exit if i > 100
         .
       Loop body
         .
     ADD R4,R4,1
     J L2
L1:
```

where **i** is held in R4. R5 is used to hold the terminating value of **i**.

Slide 186

To implement the WHILE loop **WHILE (i==j) loop body** we might have:

```
L2:  CMP R4,R5
     BNE L1
         .
       Loop body
         .
     J L2
L1:
```

where **i** is held in R4 and **j** is held in R5.

Slide 187

To implement the DO loop **`body WHILE (i==j)`** we might have:

```
L2:      .
       Loop body
           .
       CMP R4,R5
       BE L2
```

Simple change from WHILE to DO–WHILE would change the type of branch instruction.

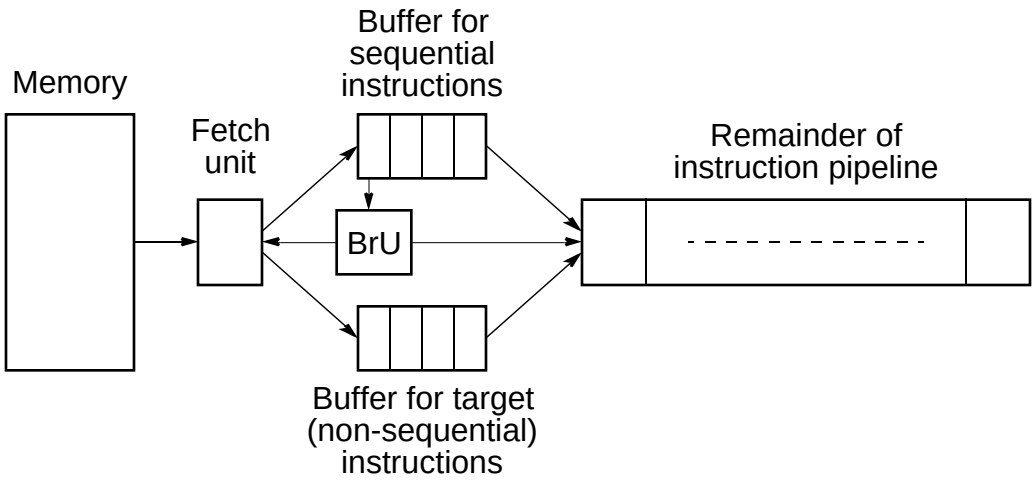
Slide 188

Strategies to reduce number of times pipeline breaks down due to conditional branch instructions

1. Instruction buffers to fetch both possible instructions.
2. Delayed branch instructions.
3. Dynamic prediction logic to fetch the most likely next instruction after a branch instruction.
4. Static prediction.

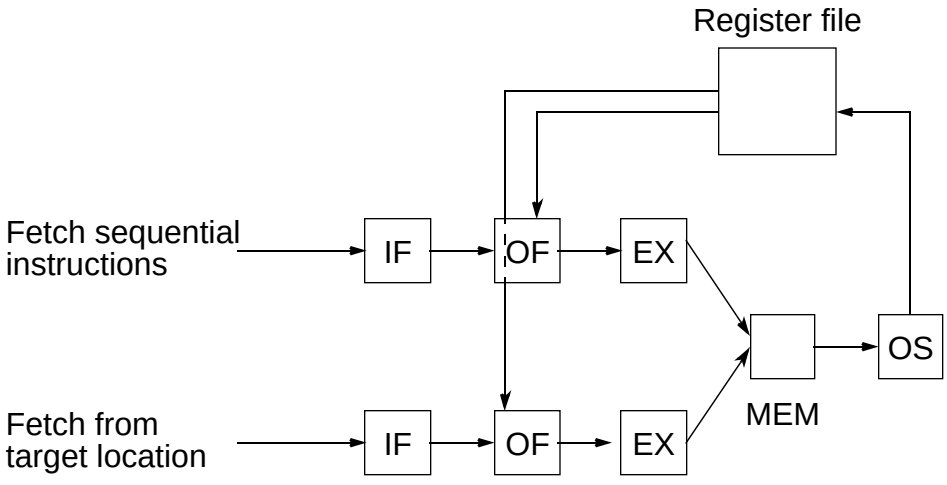
Slide 189

1. Instruction buffers

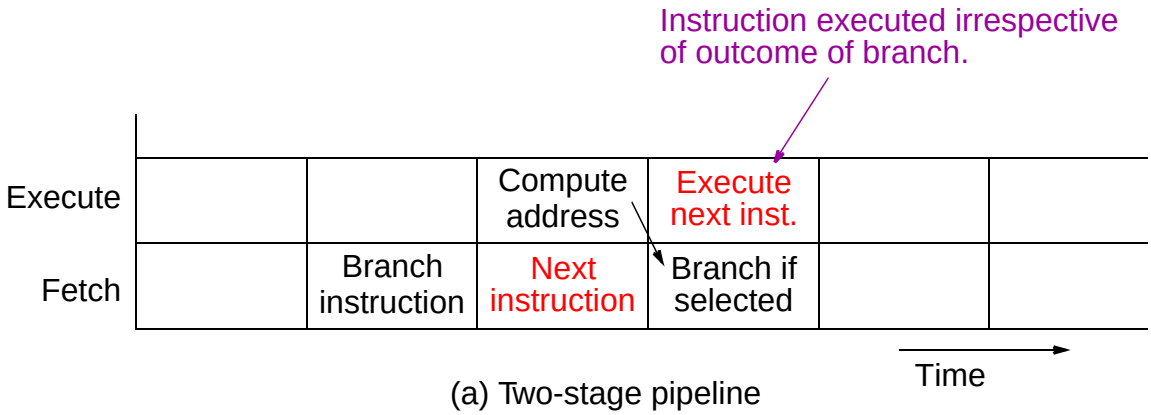


Slide 190

Replicating first stages of pipeline



2. Delayed branch instructions



(a) Two-stage pipeline

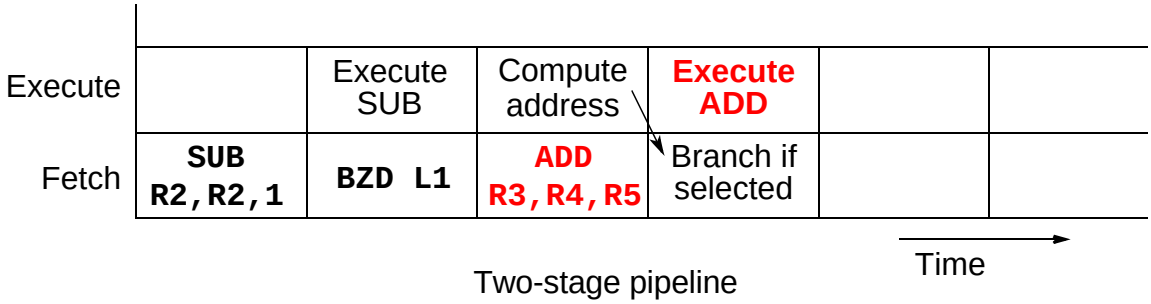
Example

```
        ADD R3,R4,R5;       ;does not affect branch
        SUB R2,R2,1
        BZ  L1
            .
L1:         .
            .
```

Move the add instruction to after the branch, i.e.:

```
        SUB  R2,R2,1
        BZD  L1
        ADD  R3,R4,R5
            .
L1:         .
            .
```

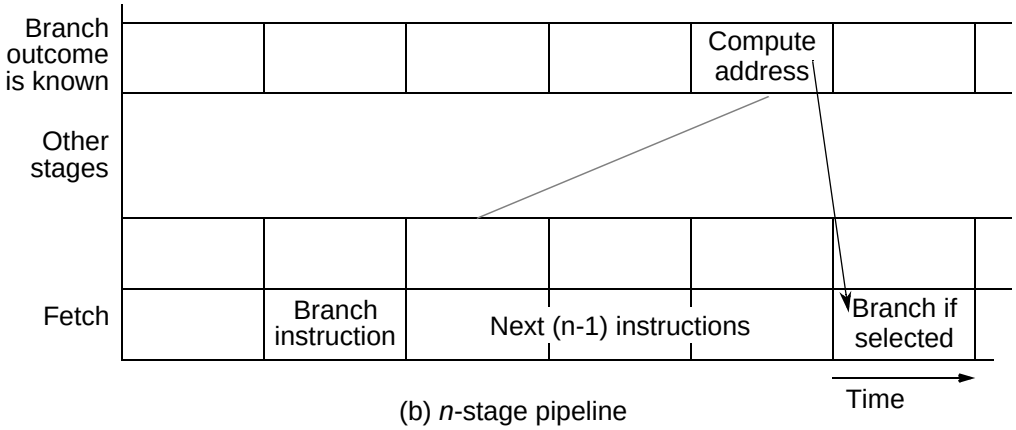
Slide 193



Two-stage pipeline

Slide 194

Could be extended for an-stage pipeline:



(b) *n*-stage pipeline

However not very effective. Compilers can find one instruction to place after branch typically 70% of time, but a second instruction 30% of the time. Delayed branch usually limited to one delay slot.

With all branches automatically delayed, use a NOP instruction whenever an instruction could not be found to place after the branch, i.e.:

```
        SUB R2,R2,1
        BZD L1
        NOP
            .
            .
L1:         .
            .
```

A compiler can easily insert these NOPs.

SUN Sparc V8 processor

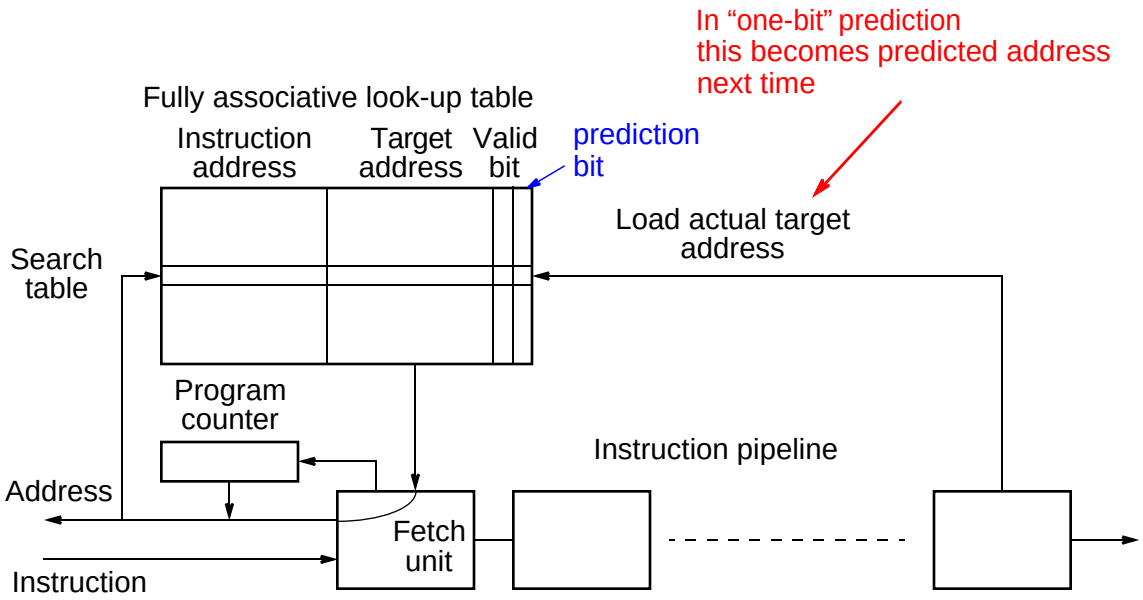
Has delayed branch instructions with an annul bit in the instruction to annul the next instruction.

Left undefined if a delayed branch instruction is placed immediately after a delayed branch instruction!!

3. Dynamic Prediction logic

Various methods of predicting the next address, mainly based upon expected repetitive usage of the branch instruction.

Usual form of prediction look-up table is a *branch history table*, also called more accurately a *branch target buffer* - similar to a cache.



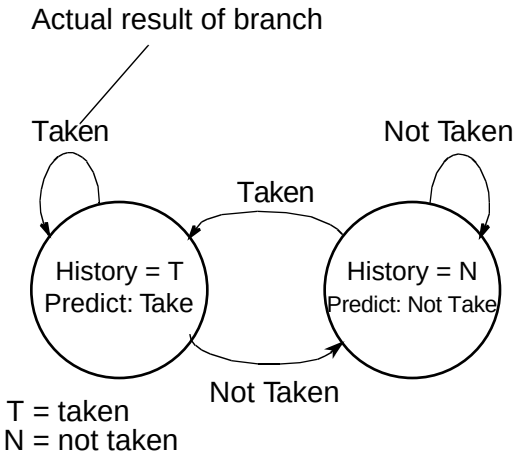
Instruction pipeline with branch target buffer

Valid bit to deal with entries not yet used.

One-bit prediction algorithm

One-bit prediction

Last branch action	Prediction
Branch taken	Take branch
Branch not taken	Not take branch



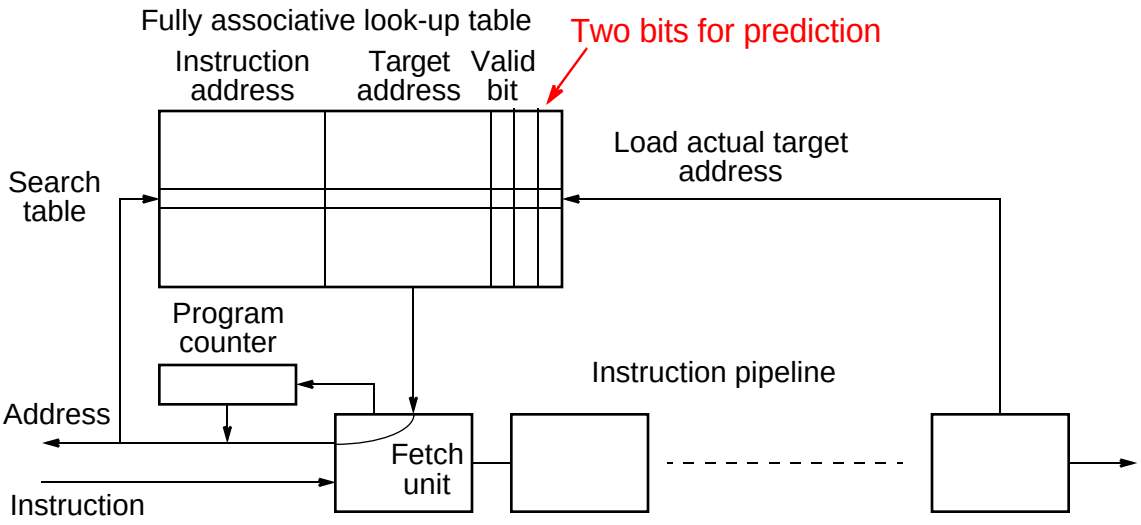
Mispredictions of one-bit prediction

One-bit prediction, we always get a misprediction when a branch instruction implementing the loop is last encountered as we then fall through the loop rather than repeat the body.

However, usually we encounter the complete loop again, i.e. the program comes back to re-execute the loop.

In that case, we get another misprediction if the prediction is updated from the last misprediction.

Hence, given a loop with *n* repetitions, 2 will be mispredicted and *n* − 2 will be predicted correctly with a single bit predictor.



Instruction pipeline with branch target buffer
Two bit prediction

(various algorithms for setting bits and intepreting them)

Two-bit prediction algorithms

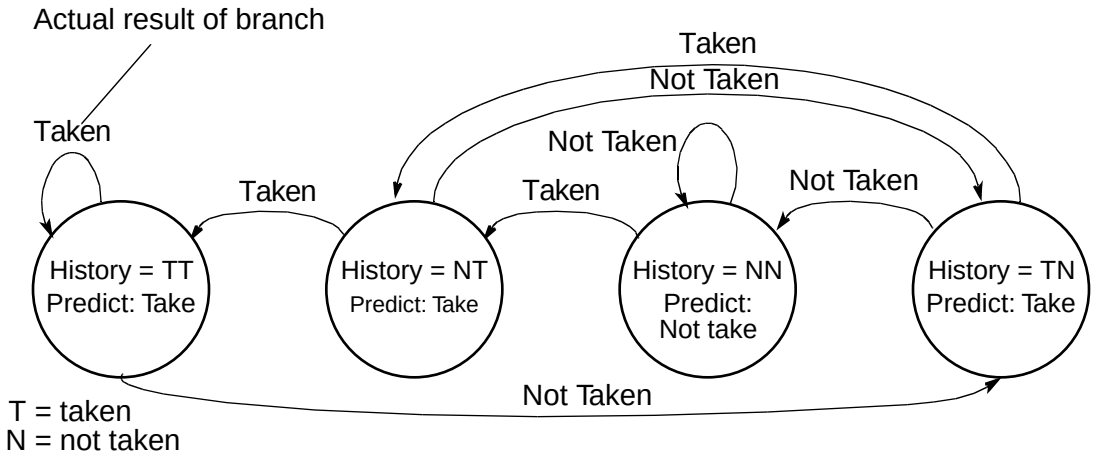
One two-bit prediction algorithm is based upon the history of *previous actions of a branch instruction* as shown below.

Two-bit prediction

History of branch actions		Prediction
Time before	Last time	
Branch taken	Branch taken	Take branch
Branch not taken	Branch taken	Take branch
Branch taken	Branch not taken	Take branch
Branch not taken	Branch not taken	Not take branch

History refers to the last two actions of a specific branch instruction. Only in one case is the prediction not to take the branch, and that is when the previous two times the branch instruction was executed, the branch was not taken; instead execution continued sequentially.

Two-bit predictor based upon history of branches

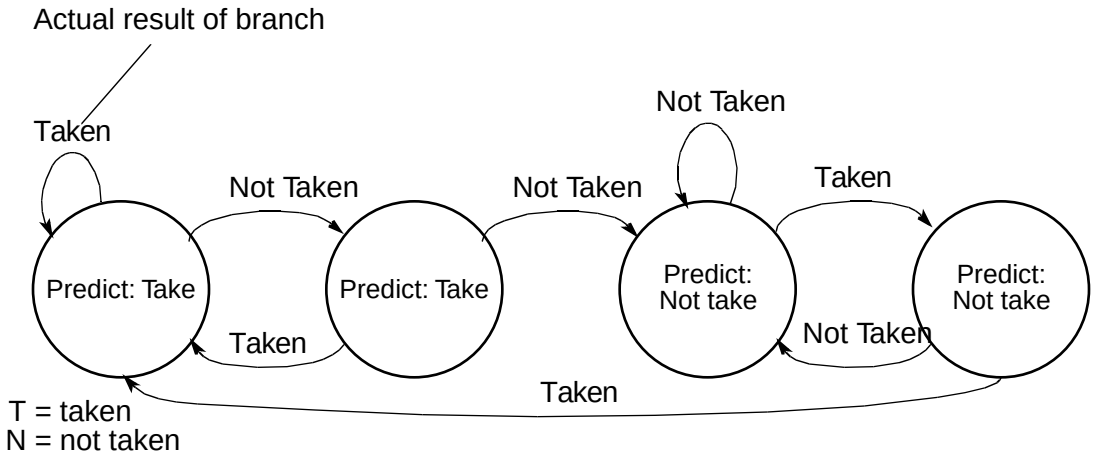


Based upon the history of the branch *predictions,* that is, the prediction is only changed after two mispredictions, as described below.

Alternative two-bit prediction

History of predictions		Prediction
Time before	Last time	
Prediction correct	Prediction correct	Keep prediction
Prediction not correct	Prediction correct	Keep prediction
Prediction correct	Prediction not correct	Keep prediction
Prediction not correct	Prediction not correct	Change prediction

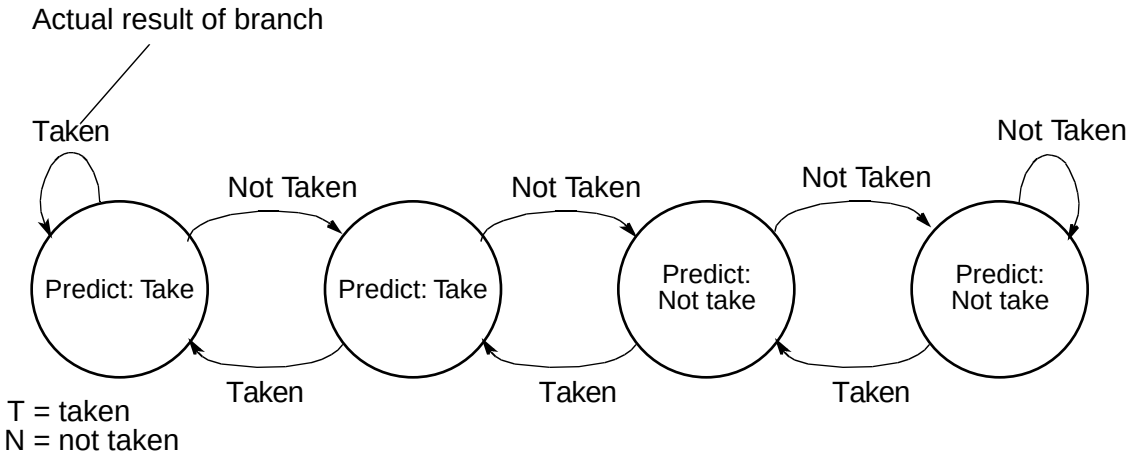
Two-bit predictor based upon history of predictions



Two-bit predictor using saturating counter

Every not taken branch causes a move to a state to the right, saturating in the rightmost state, while every taken branch causes a move to the left, saturating in the leftmost state.

The two left states cause a prediction to take the branch while the two right states cause a prediction not to take the branch. (This counter could be extended to more bits.)



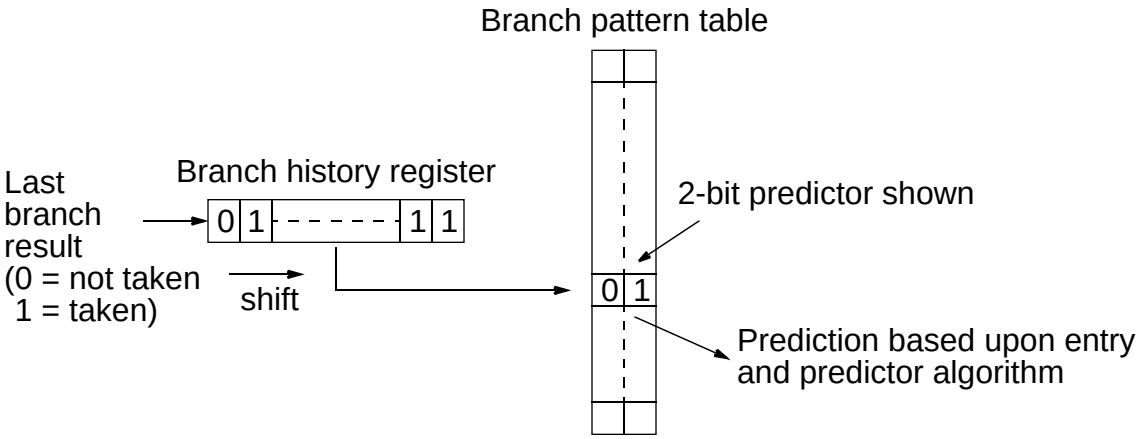
Two-bit saturation counter predictor

Correlation Predictors (two-level predictors)

The actions of branch instructions will often depend upon the actions of previous branch instructions, not necessarily the same branch instructions.

In correlation predictors, the history of branch instructions recorded as they are encountered.

For example, if the last two branch instruction executed (not necessarily the same instructions at the same addresses) all were taken branches, the history would be “taken, taken.”



Two-level (adaptive) predictor

Used in Pentium

Question

What happens to the prediction in a pipeline that fetches another branch instruction before the previous one has completed (or the same one)?

4. Static Prediction

Static prediction makes the prediction before execution.
(Note, target address cannot be computed beforehand if reg. indirect)

A very simple hardware prediction – always chooses one way (either always taken, or always not taken).

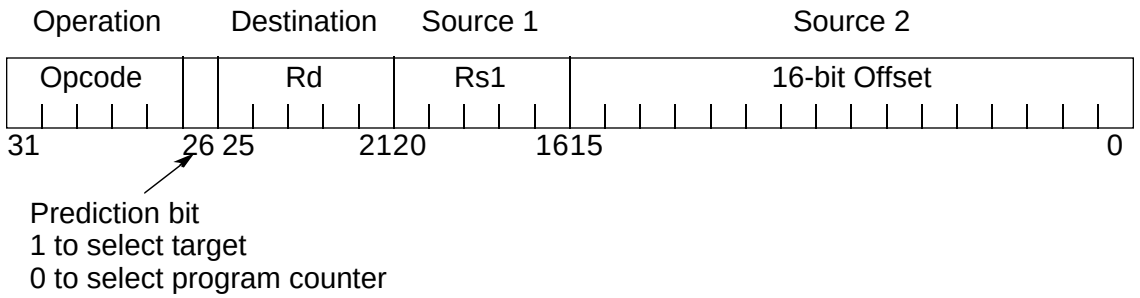
“Predict never taken” done in Motorola 68020 - essentially no prediction.

Other possibilities include “predict by branch op-code”, and “predict taken if backward branch (e.g. in loops) - predict not taken if forward branch” as done in PowerPC.

Compiler Prediction

Passed to processor by selection of branch instruction

Have a conditional branch instruction which has additional fields to indicate the likely outcome of the branch. Single bit could be provided in the instruction which is a 1 for the fetch unit to select the target address (as soon as it can), and a 0 for the fetch unit to select the next instruction.



Branch instruction format with a prediction bit

Lecture 8

Data dependencies

Describes the normal situation that the data that instructions use depend upon the data created by other instructions.

Three types of data dependency between instructions,

- true data dependency
- antidependency
- output dependency.

↕ Sometimes called name dependencies

True data dependency

Occurs when value produced by an instruction is required by a subsequent instruction. Also known as *flow dependencies* because dependency is due to flow of data in program and also called *read-after-write hazards* because reading a value after writing to it.

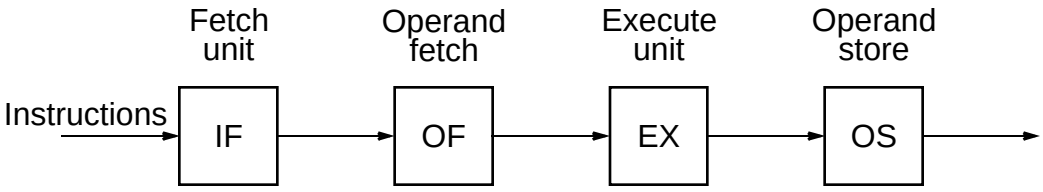
Example

```
1. ADD R3,R2,R1    ;R3 = R2 + R1
2. SUB R4,R3,1     ;R4 = R3 - 1
```

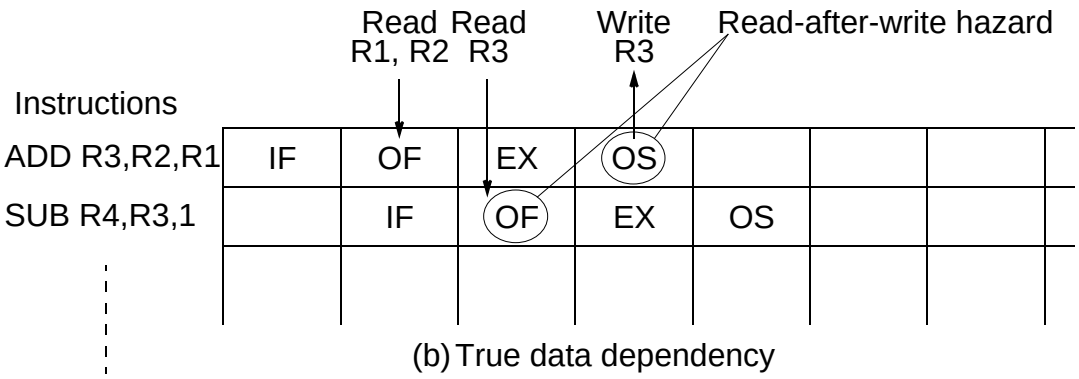
"data" dependency between instruction 1 and 2 (R3).

In general, they are the most troublesome to resolve in hardware.

True data dependency in a four-stage pipeline



(a) Stages



(b) True data dependency

Antidependency

Occurs when an instruction writes to a location which has been read by a previous instruction.

Also called an *antidependency* (and a *write-after-read hazard*).

Example

```
1. ADD R3,R2,R1    ;R3 = R2 + R1
2. SUB R2,R5,1     ;R2 = R5 - 1
```

Instruction 2 must not produce its result in R2 before instruction 1 reads R2, otherwise instruction 1 would use the value produced by instruction 2 rather than the previous value of R2.

Antidependencies in a single pipeline

In most pipelines, reading occurs in a stage before writing and an antidependency would not be a problem. Becomes a problem if the pipeline structure is such that writing can occur before reading in the pipeline, or the instructions are not processed in program order - see later.

Output dependency

Occurs when a location is written to by two instructions. Also called *write-after-write hazard*.

Example

```
1. ADD R3,R2,R1    ;R3 = R2 + R1
2. SUB R2,R3,1     ;R2 = R3 - 1
3. ADD R3,R2,R5    ;R3 = R2 + R5
```

Instruction 1 must produce its result in R3 before instruction 3 produces its result in R3 otherwise instruction 2 might use the wrong value of R3.

Output dependencies in a single pipelne

Again the dependency would not be significant if all instructions write at the same time in the pipeline and instructions are processed in program order. Output dependencies are a form of resource conflict, because the register in question, R3, is accessed by two instructions. The register is being reused. Consequently, the use of another register in the instruction would eliminate the potential problem.

Slide 221

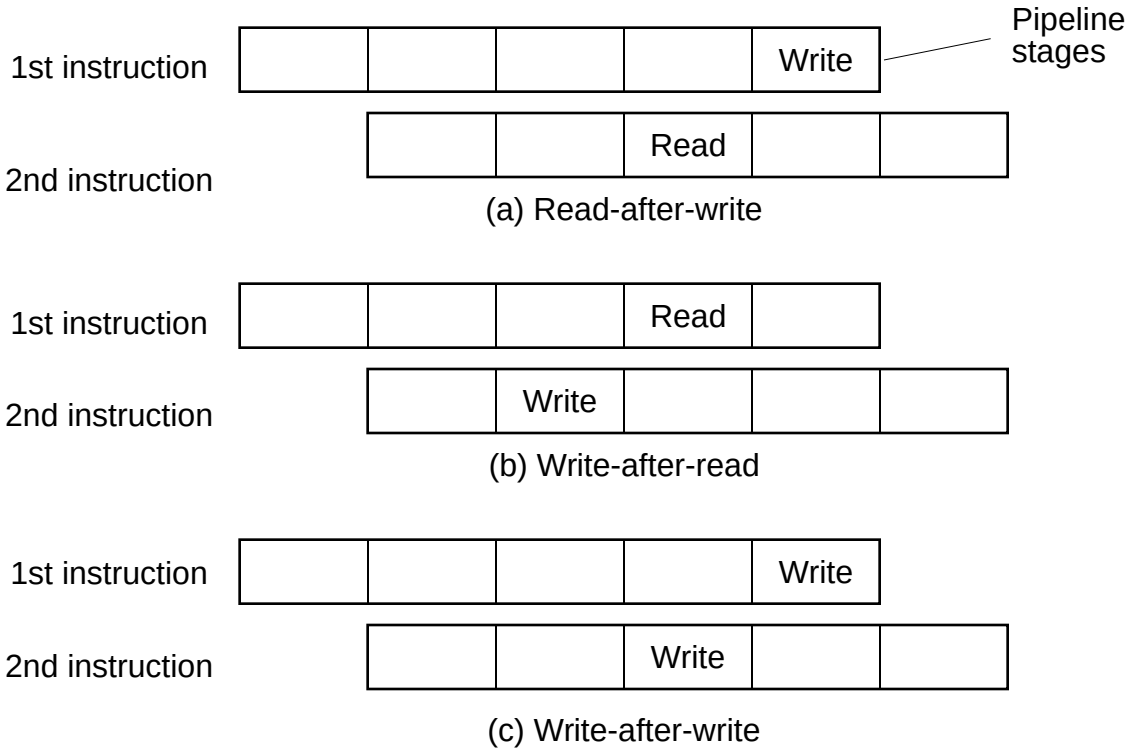
Detecting hazards

Can be detected by considering read and write operations on specific locations accessible by the instructions:

- A *read-after-write* hazard exists if read operation occurs before previous write operation has been completed, and hence read operation would obtain incorrect value (a value not yet updated).
- A *write-after-read* hazard exists when write operation occurs before previous read operation has had time to complete, and again the read operation would obtain an incorrect value (a prematurely updated value).
- A *write-after-write* hazard exists if there are two write operations upon a location such that the second write operation in the pipeline completes before the first.

Read-after-read hazards, in which read operations occur out of order, do not normally cause incorrect results.

Slide 222



Read/write hazards

Mathematical Conditions for Hazard

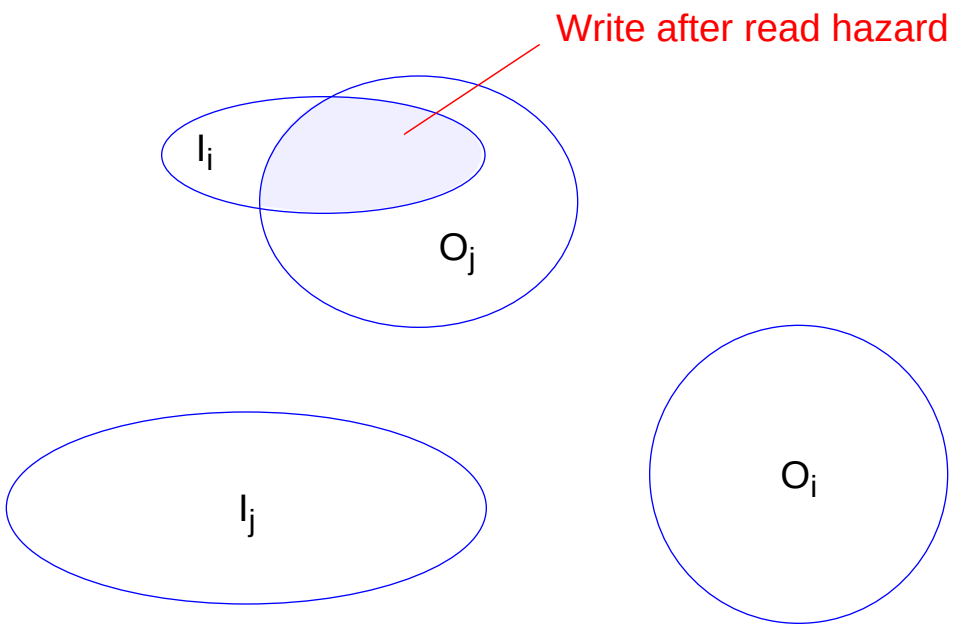
(Berstein's Conditions)

O(*i*) indicates set of (output) locations altered by instruction *i*; I(*i*) indicatesset of (input) locations read by instruction *i*, and φ indicates an empty set.

A potential hazard exists between instruction *i* and subsequent instruction *j* when at least one of the following conditions fails:

Null set
i.e. left are disjoint sets

For read-after-write	O(*i*)	∩	I(*j*)	=	φ
For write-after-read	I(*i*)	∩	O(*j*)	=	φ
For write-after-write	O(*i*)	∩	O(*j*)	=	φ



Example

Suppose we have code sequence:

```
1. ADD R3,R2,R1   ;R3 = R2 + R1
2. SUB R5,R1,1    ;R5 = R1 - 1
```

entering the pipeline. Using these conditions, we get:

O(1) = (R3), O(2) = (R5)
I(1) = (R1,R2)
I(2) = (R1).

The conditions: **(R3)** $\cap$ **(R1) =** φ, **(R2, R1)** $\cap$ **(R5) =** φ, **(R3)** $\cap$ **(R5) =** φ, are satisfied and there are no hazards

Berstein's Conditions can be extended to cover more than two instructions.

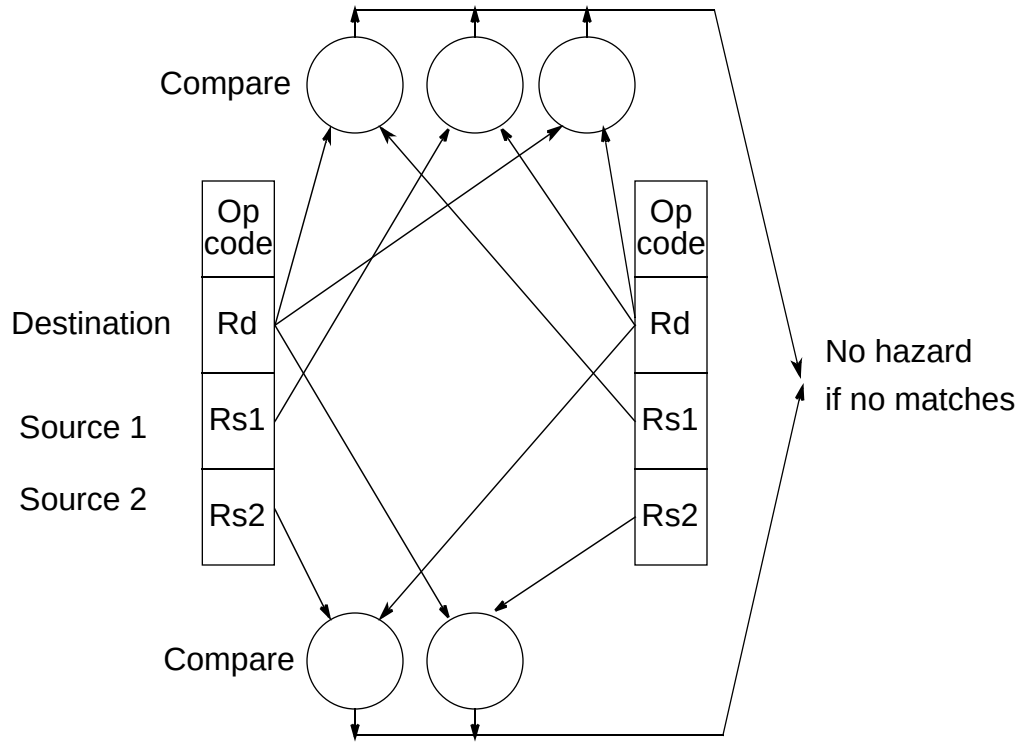
Number of hazard conditions to be checked becomes quite large for a long pipeline having many partially completed instructions.

Satisfying conditions are sufficient but not necessary in mathematical sense. May be that in a particular pipeline a hazard does not cause a problem.

Direct hardware method of checking Berstein's conditions

Do logical comparisons between the source and destination IDs of instructions that have been fetched and held an instruction buffer prior to execution, or held in the pipeline latches during execution.

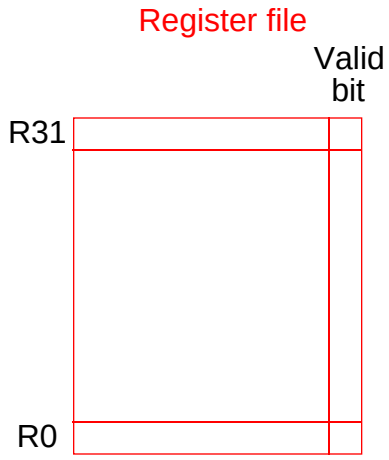
Checking hazards between two instructions in pipeline



Although previous method can be extended to any number of instructions, it is complex and a much simpler method that is usually sufficient is as follows:

Pipeline interlock using register valid bits

Associate a 1-bit flag (valid bit) with each operand register. Flag indicates whether a valid result exists in register, say 0 for not valid and 1 for valid.



Resetting valid bit (invalid)

A fetched instruction which will write to the register examines the valid bit. If the valid bit is 1, it is reset to 0 (if not already a 0) to show that the value will be changed.

(We shall show later that the instruction must stall if its destination register is already set invalid.)

Slide 231

Setting bit (valid)

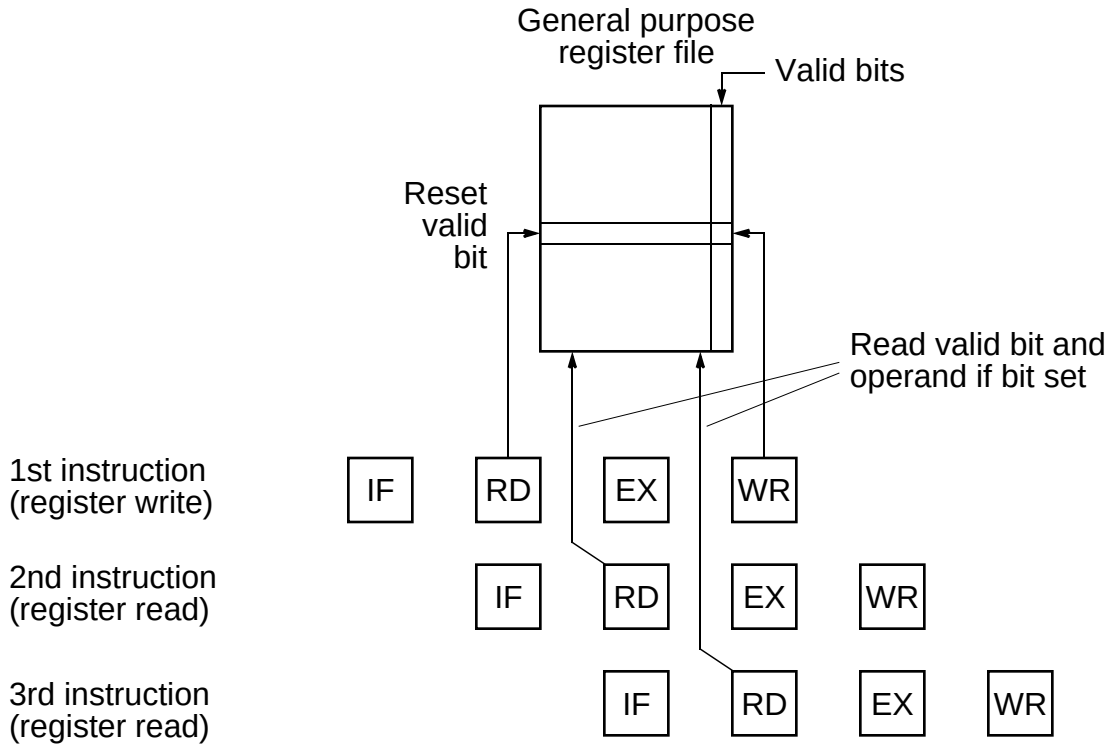
When the instruction has produced the value, it loads the register and sets the valid bit to 1, letting other instructions have access to the register.

Slide 232

Reading valid bit

Any subsequent instruction which wants to read register operands has to wait until the registers valid bits have been set before reading the operands.

Slide 233



Register read/write hazard detection using valid bits (IF, instruction fetch; RD, read operand; EX, execute phase; WR write operand)

Slide 234

Example

Suppose the instruction sequence is:

1. `ADD R3,R4,4`
2. `SUB R5,R3,8`
3. `SUB R6,R3,12`

Read-after-write hazard between instr. 1 and instr. 2 (R3).
Read-after-write hazard between instr. 1 and instr. 3 (again R3).

In this case, sufficient to reset valid bit of R3 register to be altered during stage 2 of instr. 1 in preparation for setting it in stage 4.

Both instructions 2 and 3 must examine valid bit of their source registers prior to reading contents of registers, and will hesitate if they cannot proceed.

Caution

The valid bit approach has the potential of detecting all hazards, but write-after-write (output) hazards need special care.

Example

Suppose the sequence is:

1. `ADD R3,R4,R1`
2. `SUB R3,R4,R2`
3. `SUB R5,R3,R2`

(very unlikely sequence, but poor compiler might create such redundant code).

Write-after-write dependency between instr. 1 and 2. Instr. 1 will reset valid bit of R3 in preparation to altering its value. Instr. 2 immediately behind it would also reset valid bit of R3 because it too will alter its value, but will find valid bit already reset. If instr. 2 were to be allowed to continue, instr. 3 immediately behind instr. 2 would only wait for the valid bit to be set, which would first occur when instr. 1 writes to R3. Instr. 3 would get value generated by instr. 1, rather than value generated by instr. 2 as called for in program sequence.

Slide 237

Correct algorithm for resetting valid bit

WHILE destination register valid bit = 0 wait (pipeline stalls), else reset destination register valid bit to 0 and proceed.

Slide 238

Forwarding

Refers to passing result of one instruction directly to another instruction to eliminate use of intermediate storage locations.

Can be applied at compiler level to eliminate unnecessary references to memory locations by forwarding values through registers rather than through memory locations. Would generally increase speed, as accesses to processor registers faster than to memory locations.

Forwarding can also be applied at hardware level to eliminate pipeline cycles for reading registers updated in a previous pipeline stage. Eliminates register accesses by using faster data paths.

Internal forwarding

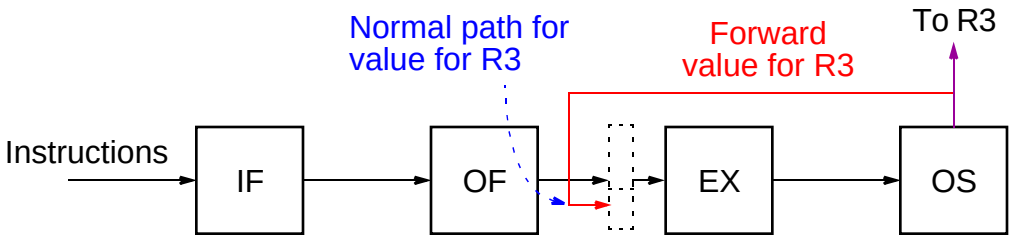
Internal forwarding is hardware forwarding implemented by processor registers or data paths not visible to the programmer.

Example

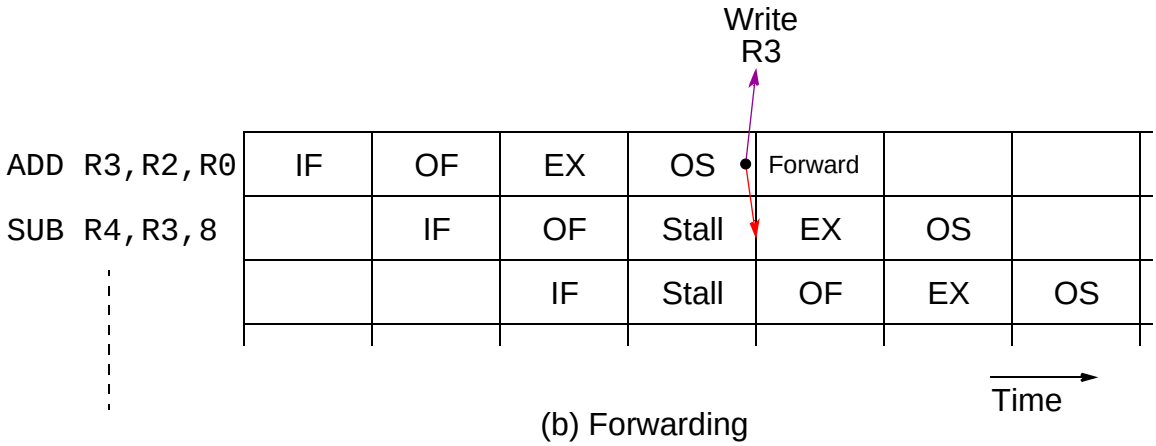
```
ADD R3,R2,R0
SUB R4,R3,8
```

Subtract instruction requires contents of R3, which is generated by the add instruction. The instruction will be stalled in the operand fetch unit waiting for the value of R3 to be come valid.

Internal forwarding forwards the value being stored in R3 by the operand store unit directly to the execute unit.



(a) Stages

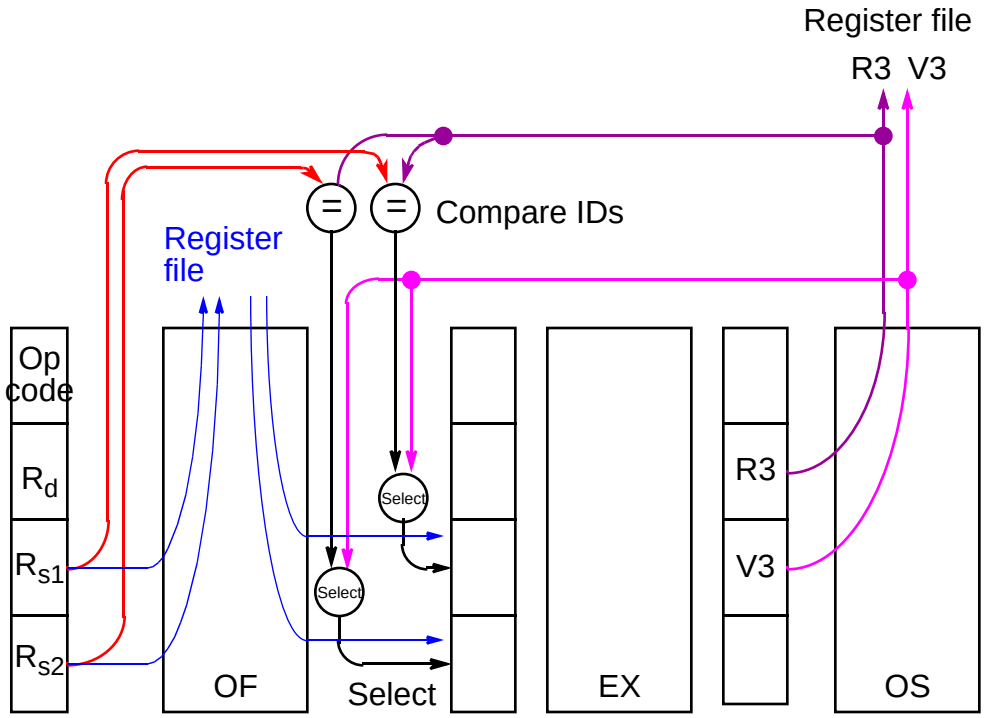


(b) Forwarding

Internal forwarding in a four stage pipeline

Internal forwarding in a four stage pipeline

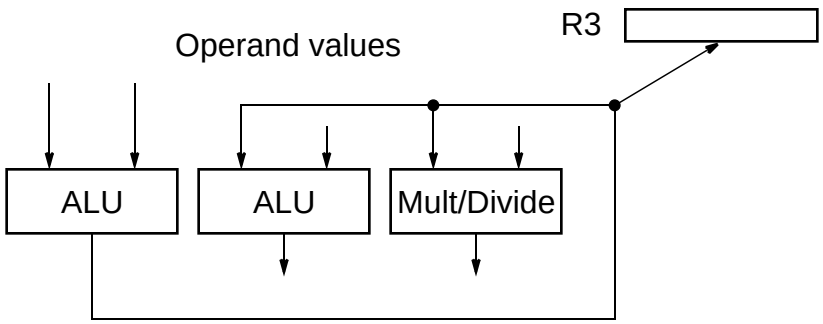
- more details



Forwarding using multiple functional units

Concept of forwarding can be taken further by recognizing that instructions can execute as soon as the operands they require become available, and not before.

Each instruction produces a result which needs to be forwarded to all subsequent instructions that are waiting for this particular result.



We shall see later in higher performance processors.