Review of Stored Program Concept

Stored Program Computer

Has a list of instructions held in a memory (stored program) which define the actions of computer.

Consists of:

- Memory
- Processor
- Input circuits and devices
- Output circuits and devices

Processor fetches (machine) instructions from memory and performs actions defined.

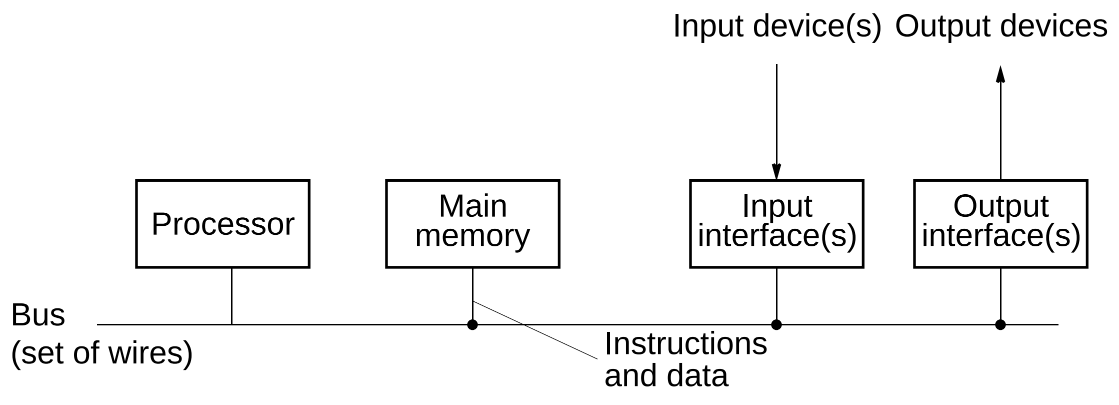
Babbage

Concept first proposed by Babbage in the 1800’s - his machine was mechanical but had the main parts of a modern computer; an arithmetic unit and controller (processor), memory (punched cards) - never completed because the mechanical complexity (gears etc).

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. multiple and dedicated buses).



Main memory

Set of storage locations

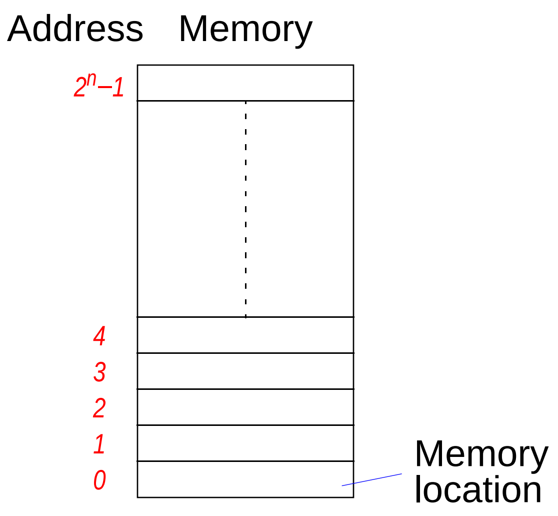
Each location given a unique address (a binary number starting from zero)

Each "addressable" location holds fixed number of bits (binary digits) - normally 8 bits. WHY?

Eight bits called a byte.

Any location can be accessed at high speed in any order (random access memory).

From CSCI 3182



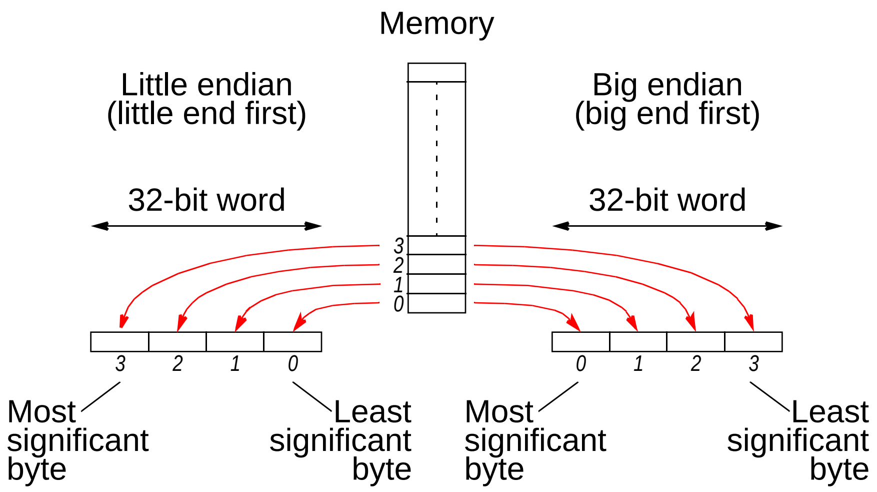
2^n locations require n-bit address

Memory used to hold machine instructions and data.

If more than 8 bits needed, consecutive locations used.

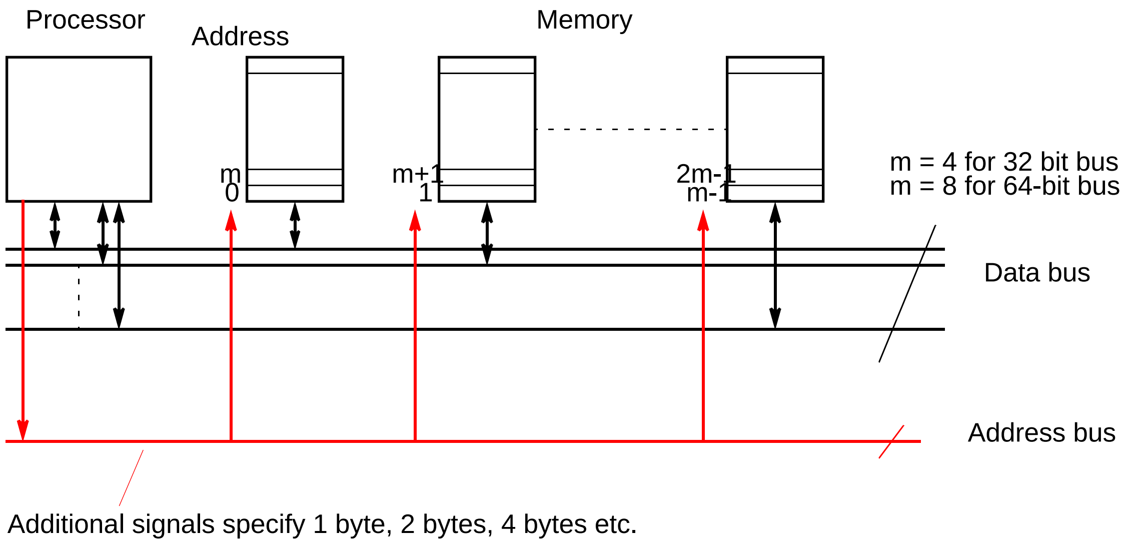
Then address given by address of first location.

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

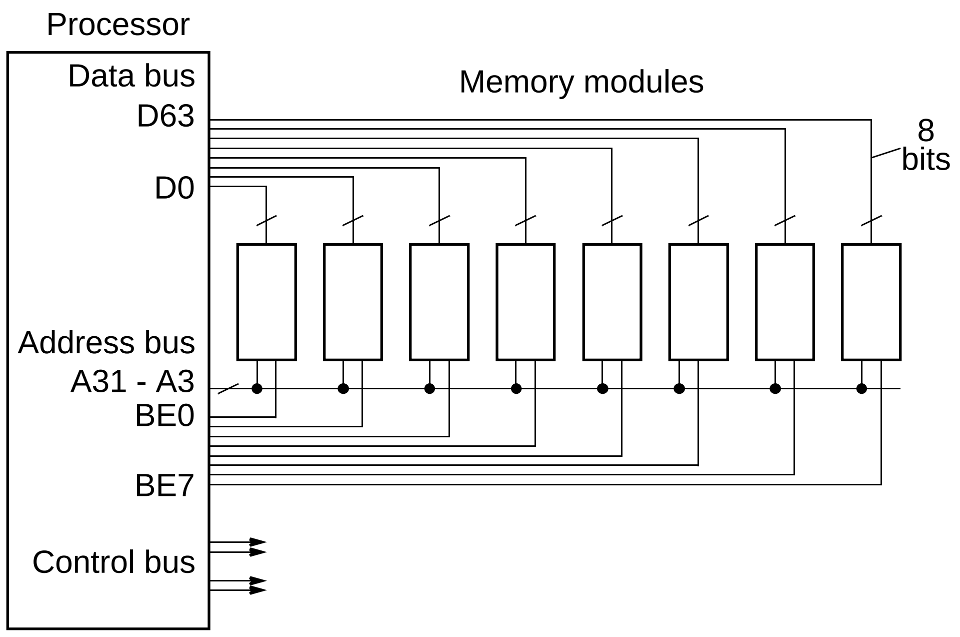


Data path between processor and memory

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



Example of a 64-bit processor bus (Pentium)



Machine Instructions

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

Various possible formats.

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

Op-code	Identifies operands and result location

The rest of the instruction specifies the locations of the numbers (operands) to be used in the operation and where the 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.

Op-code encoding

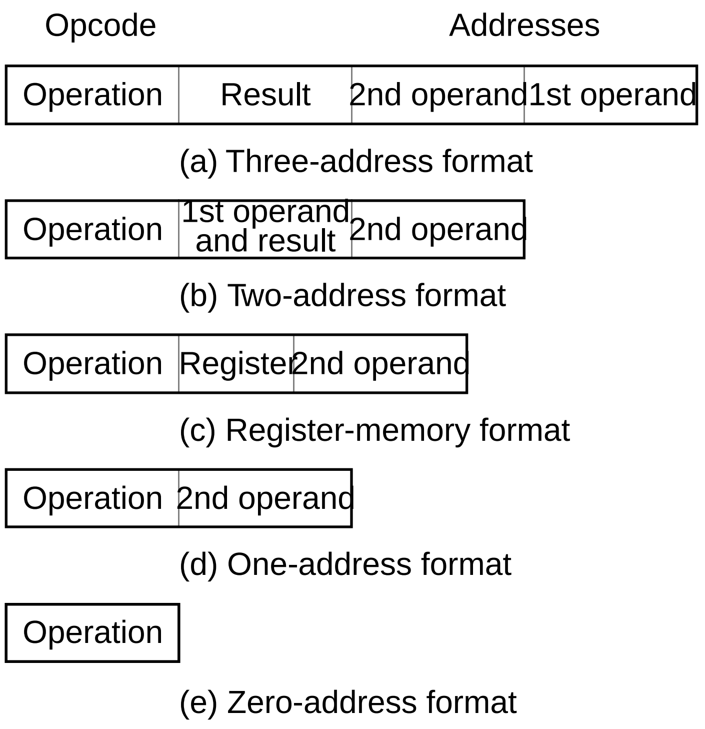
Suppose there were 100 different operations, add subtract, multiply, divide, ... 7 bits would be sufficient ($2^6 <= 100 < 2^7$). Could allocate one pattern for each operation:

```
                                          op-code
ADD (Usually abbreviated to ADD)0000000
SUBTRACT (Usually abbreviated to SUB)0000001
MULTIPLY (Usually abbreviated to MUL)0000010
DIVIDE (Usually abbreviated to DIV0000011
.            .
```

Sometimes a more complex encoding is used, say, the first bits specify a class of operation and the rest of the op-code specifies the operation within

Instruction Formats

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



Identifying the operands and result location

Various methods, known generally as the addressing modes.

Principal addressing modes

Absolute addressing - Operand in memory and thememory 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. The address of the operand location is held in a register which is specified in instruction.

Register Indirect Addressing plus Displacement - Similar to register indirect addressing except a displacement or offset held in the instruction is added to register contents to form the effective address.

Implied Addressing - Some operations have an implicit location for the operand and its address need not be specified.

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

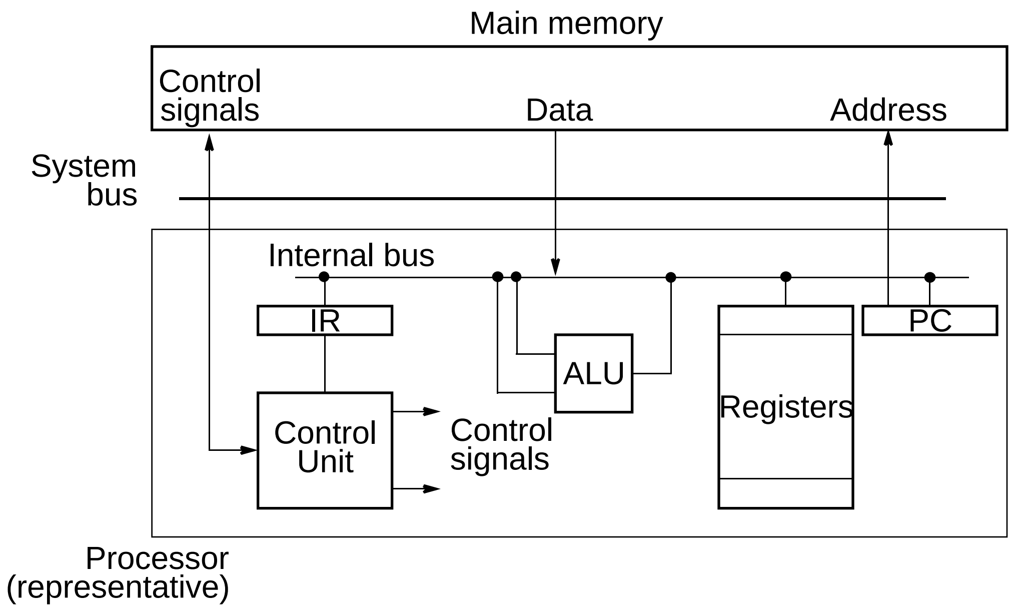
Processor

Reads ("fetches") so-called machine instructions for the memory which constitutes the executable program, and performance the actions specified ("executes" them).

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

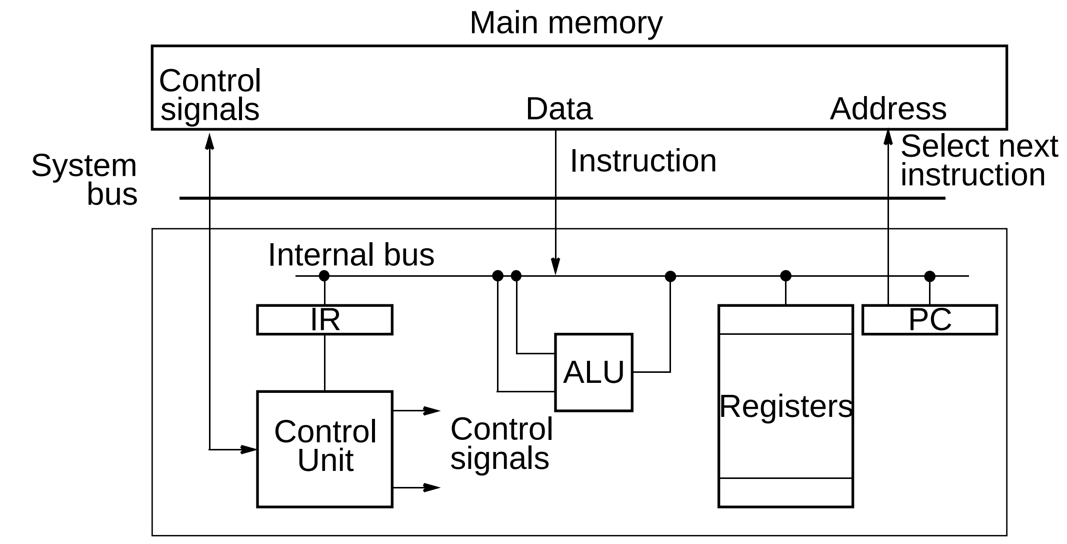
A very simple processor

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

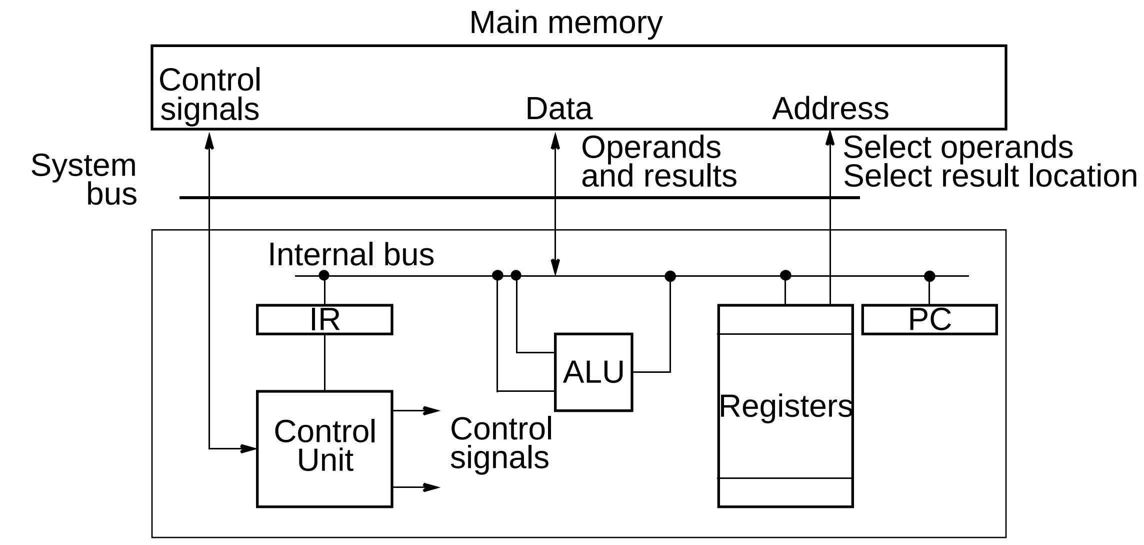


A very simple processor will operate in two phases, a fetch cycle to fetch an instruction and an execute cycle to execute the fetched instruction. These two cycles are repeated as the program is executed. Usually however processors will attempt to fetch the next instruction before the previous one has been fully executed.

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



(a) Fetch cycle



(b) Execute cycle

Summary

We can identify the main operating characteristics of the stored program computer as follows:

1. Only elementary operations are performed (e.g., arithmetic addition, logical operations).
2. The user (programmer) or compiler selects operations to perform the required computation.
3. Encoded operations are stored in a memory.
4. Strict sequential execution of stored instructions occurs (unless otherwise directed).
5. Data may also be stored in the same memory.

Characterizing Performance

MIPs

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

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

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, the SPECint92 and SPECfp92 UNIX benchmarks.

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 actually be less than one if processor is capable of executing more than one instruction simultaneously (as most processors can since the mid 1990’s).

Improvements in performance

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

Development of microprocessor families

Processors developed in "families" such that processors would be able to execute programs of earlier processors.

Eight-bit microprocessors, that is, processors that can operate upon and perform arithmetic on 8-bit numbers directly, in the mid- 1970s, typified by the Intel 8080, Motorola MC6800 and Zilog Z-80.

Sixteen-bit microprocessors towards the end of the 1970s, e.g., Intel 8086 and Motorola MC68000, both introduced in 1978.

Thirty-two bit processors appeared in 1980s (e.g., Intel 386, Motorola MC68020, and MC68030. Intel 486 and Motorola MC68040 continued the trend of adding facilities within the chip (floating point units).

Development of microprocessor families cont.

Superscalar processors in early 1990s - more than one instruction can be executed in each cycle, e.g. Intel Pentium and Pentium Pro, superscalar versions of 486. Pentium II introduced in 1996, extra instructions for multimedia applications (MMX technology).

Sixty-four bit processors in mid 1990s.

Also Intel pursuing a design in their Pentium III in which instructions are packaged into groups for simultaneous execution.

Reduced Instruction Set Computer (RISC emerged in early 1980s. Instruction set carefully chosen (perhaps less than 100 instructions) and a few addressing modes (perhaps 2–5) – leads to processor that can operate faster - now basis of all processors.

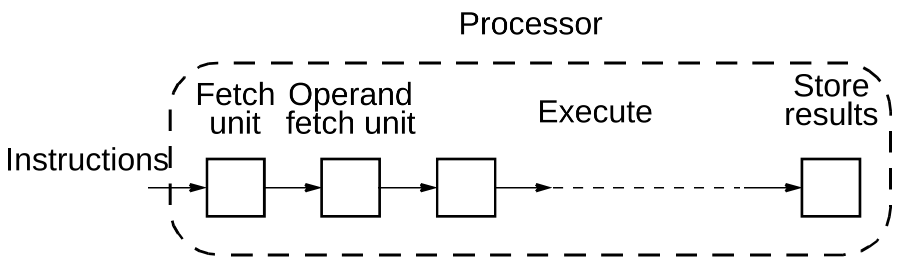
Pipelined processor design

A basic techniques to improve the performance - pipeline technique, always applied in high performance systems.

The operation of the processor can be divided into a number of steps, e.g.:

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

or more steps.



Memory hierarchy

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

Main memory – random access memory

Secondary memory – not random access but not volatile

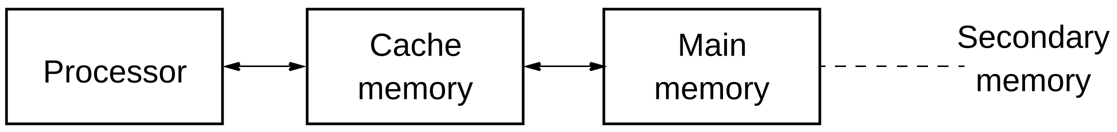
Usually being based upon magnetic technology.

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 – a method of hiding the memory hierarchy.

Cache

High speed memory introduced between the processor and main memory:



Instruction Set

The instructions that a processor can execute.

Complex Instruction Set Computers (CISC)

Early computers by necessity had small instruction sets. However during the development of computers in the 1960's and 1970's, there was a 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.

Reduced instruction set computer (RISC)

The following issues originally lead to RISC concept:

1. The effect of the inclusion of complex instructions.
2. The best use of transistors in VLSI implementation.
3. The overhead of microcode.
4. The use of compilers.

The basic questions asked were "What effect on the design of the processor does all these extra instruction have on the operation of the processor?" and "Do the extra instructions indeed increase the speed of the system?"

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.

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 has been 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?

Microcode

Factor leading to original RISC concept was changing memory technology. CISCs often rely heavily on microprogramming (microcode) in which fast control memory inside the processor holds microinstructions specifying the steps to perform for each machine instruction.

Microprogramming first used at time when main memory was based upon magnetic core stores and faster read-only control memory could be provided inside the control unit.

With the 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.

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.

A key part of the 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.

RISC examples

IBM 801

Designed 1975–79 and publicly reported in 1982. Establishes many of the 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.

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.

. •

Table 1.1 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 the RISC I/II and MIPS led to commercial RISC processors:

SUN Sparc processor is derived from the Berkeley RISC II processor.

MIPS Computer System Corporation was 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 is maybe first RISC produced by major CISC microprocessor manufacturer.

Later RISCs also incorporated superscalar operation (executing more than one instruction in one clock cycle). Examples include IBM RS 6000, DEC Alpha family and PowerPC family.

Examples of 64-bit superscalar processors include the Alpha 21164, MIPS 10 000, PowerPC 620 and the UltraSparc.

RISC Instruction Set Design

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.*

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

Thirty-two integer registers are commonly chosen. Compromise between providing compilers with enough registers, and having too many registers to save beforecontext 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. Often R0 although it could be any register. (DEC Alpha processor, for example, uses R31 to hold zero.)

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 are provided for holding floating point numbers, say F0 to F31.

The numbers 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).

One register could be used to hold zero in floating point format

Operand and instruction size

Closely linked to allowable complexity of fabrication technology. Thirty-two bits provide reasonable precision for integers, but by the 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).

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

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.

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, .

Present TREND

64 bits, provides for main memory up to 2^{64} bytes (??? gigabytes). This size of main memory unlikely to be practical for many years (if ever) but ensure longevity of design.

Instruction formats

Register-register instructions

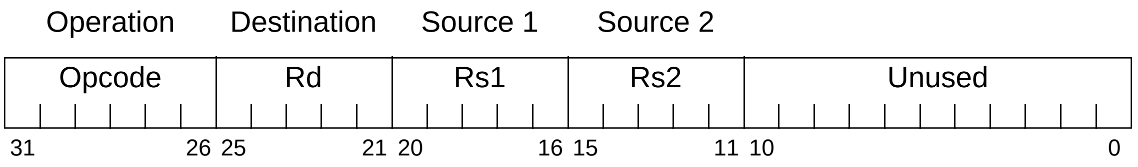
Example:

```
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)
```

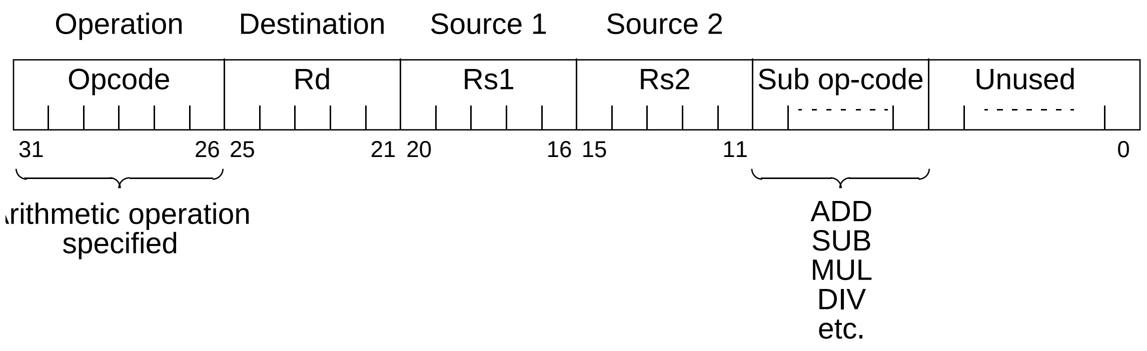
so that a specific register move instruction unnecessary.



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

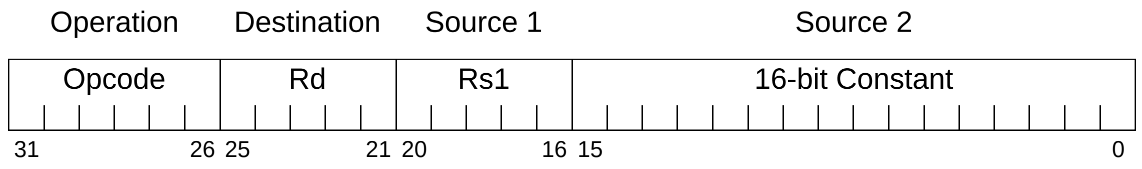
Register-register-constant format

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,R3,16    ;R2 = R3 + 16
```

Constant held in instruction would be sign-extended to the number of bits of the register.



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

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.

Example

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

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

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

Register-memory format

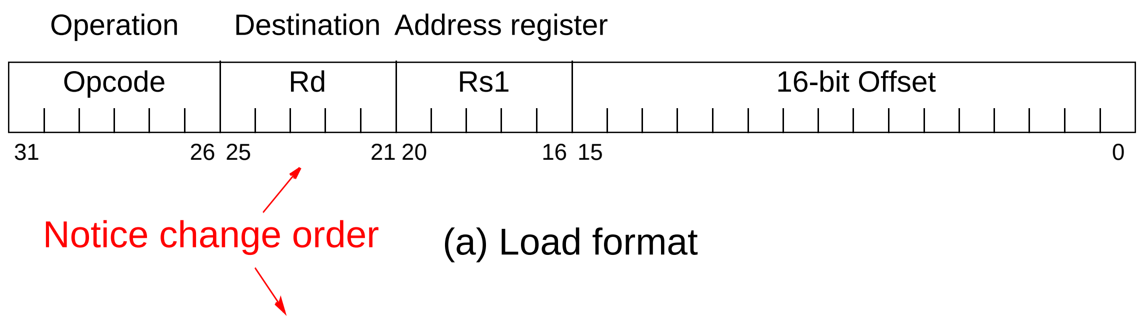
For loading registers from memory locations and storing registers in memory locations.

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

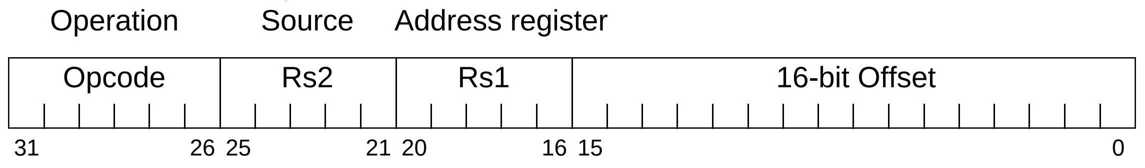
Register-memory format

Example:

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

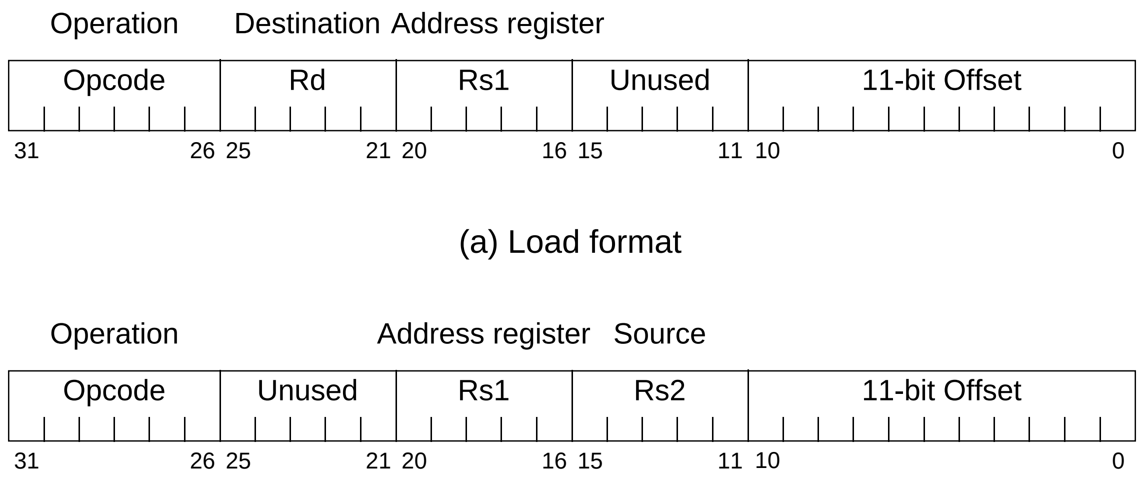


(a) Load format



(b) Store format

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



(a) Load format

(b) Store format

Load/store instruction formats (using R-R-I format) - version 2

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 try to provide a unique machine instruction for this IF statement because of the 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:
```

There is more than one way of creating above IF statement.

There is more than one way of implementing:

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

as one or more 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.

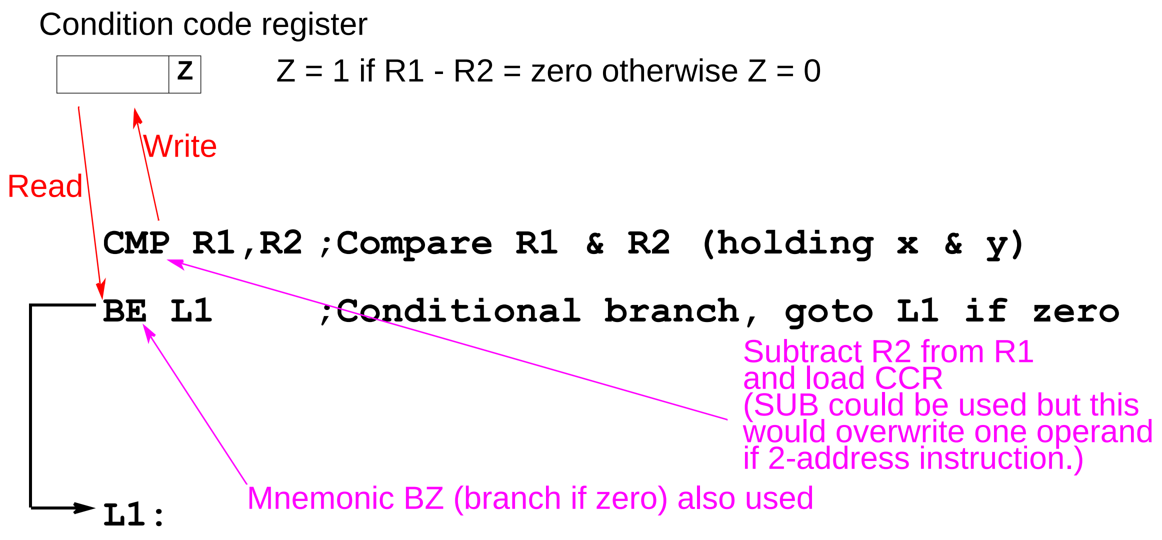
The result of test of the first instruction is stored in a so-called condition code register (CCR) for the second intruction to read.

Example

The if statement:

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

could be implemented by a sequence of two instructions:


Condition code register
Z
Z = 1 if R1 - R2 = zero otherwise Z = 0
Write
Read
Subtract R2 from R1 and load CCR (SUB could be used but this would overwrite one operand if 2-address instruction.)
Mnemonic BZ (branch if zero) also used

```
CMP R1,R2 ;Compare R1 & R2 (holding x & y)
BE L1     ;Conditional branch, goto L1 if zero

L1:
```

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 usally exist include carry (C), and overflow (O)

Conditional Branch Instructions

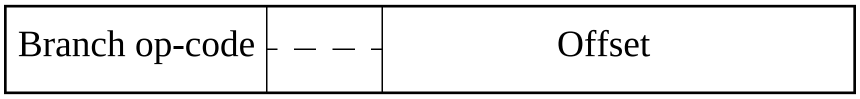
Mnemonic	Condition	C notation	Flags checked*
BL	Branch if less than	<	S
BG	Branch if greater than	>	$\overline{S} + \overline{Z}$
BGE	Branch if greater or equal to	>=	$\overline{S}$
BLE	Branch if less or equal to	<=	$S + Z$
BE	Branch if equal	==	Z
BNE	Branch if not equa1	!=	$\overline{Z}$

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

Specifying Target Location (L1)
PC-Relative Addressing

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

PC-Relative Addressing - the number of locations from the address of the present (or next) instruction is held in the instruction as an offset.



Offset is added to the program counter to obtain the effective address.

Also good programming practice to limit sequence changes to a short distance from the 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.)

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
```

The problem with CCR approach

- Requires the first action (compare) to be perfomed completely before the second action (branch) can be started.
- The two instruction must be performed sequentially and generally be next to each other.
- Hence, limits the processor from executing instructions simultaneously or not in program order (which can improve performance).

Avoiding use of condition code register

Combined test 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:

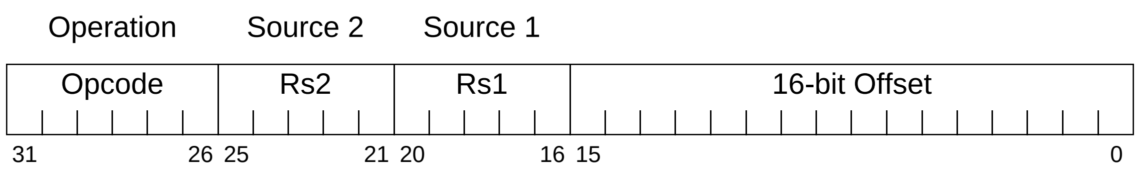
Combined test and branch instructions

```
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).

Strictly not all are necessary (Which ones form a sufficient subset?)

Combined Test 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

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
BEQNZ R3,L1  ;Branch to L1 if R3 ≠ 0
```

though in our case it is easy to accomplish with R0, i.e.:

```
BEQ 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.

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

For example, the C loop:

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

can be reduced to

```
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

The jump instruction causes an unconditional change of execution sequence to a new location.

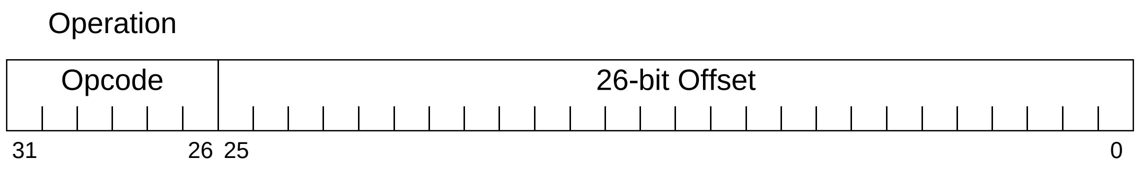
Necessary to implement more complicated IF constructs, FOR, and WHILE loops.

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

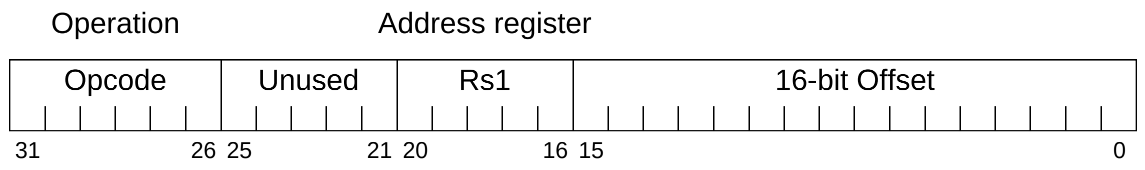
An “address register” specified in the instruction holding the address of the location holding the next instruction to be executed. Used, for example, to implement SWITCH/CASE statements in a high level language. May also be necessary for procedure return operations (see later).

```
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 is specified as an absolute address, rather than as a relative address.



Jump instruction format with register indirect addressing

Procedure 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 parameters 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.

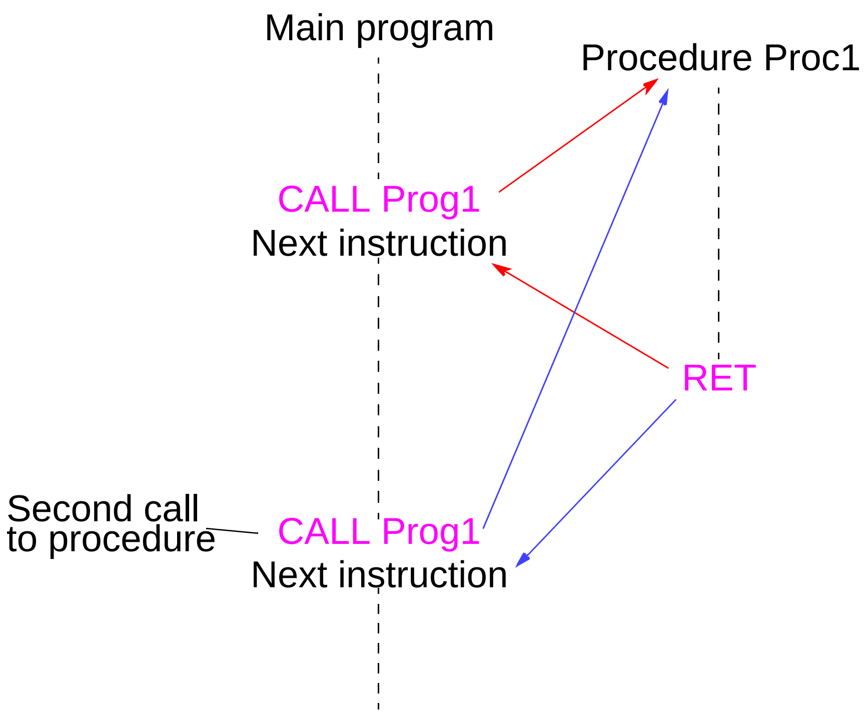
Methods to implement Procedures

CALL/RET instructions

Special machine instructions for both procedural call and procedural return in the 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.



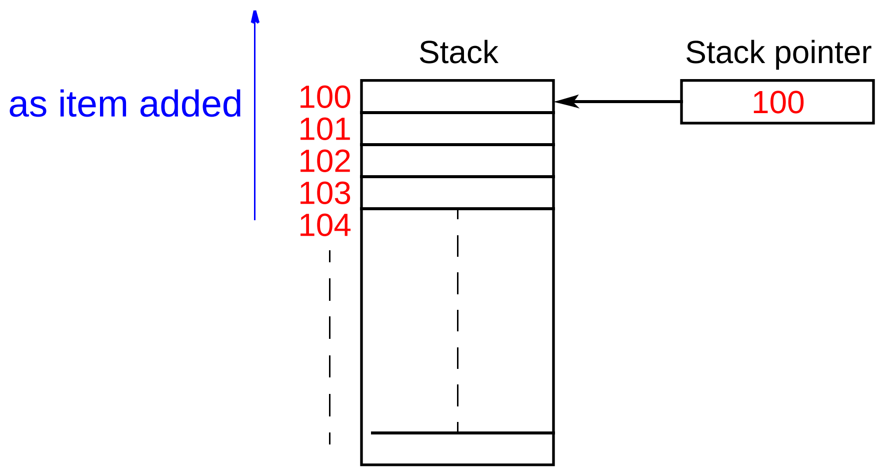
Procedure calls using CALL and RET instruction

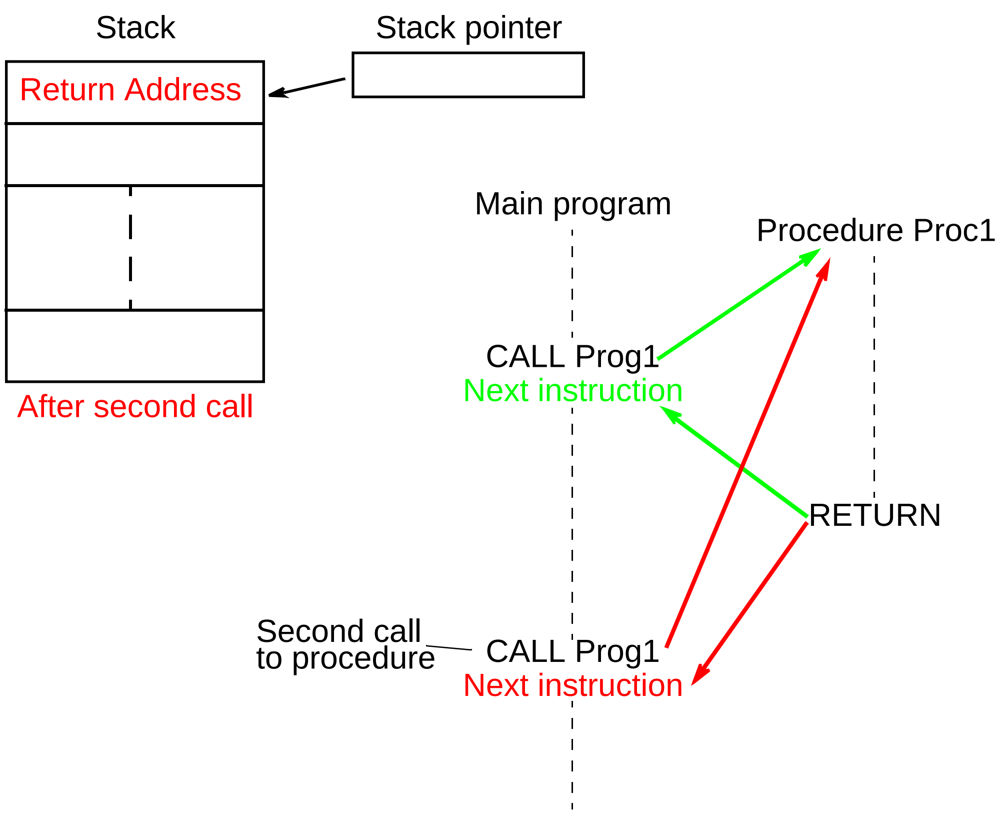
Stacks

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 have been used because they allow almost limitless nesting and recursion of procedures.

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

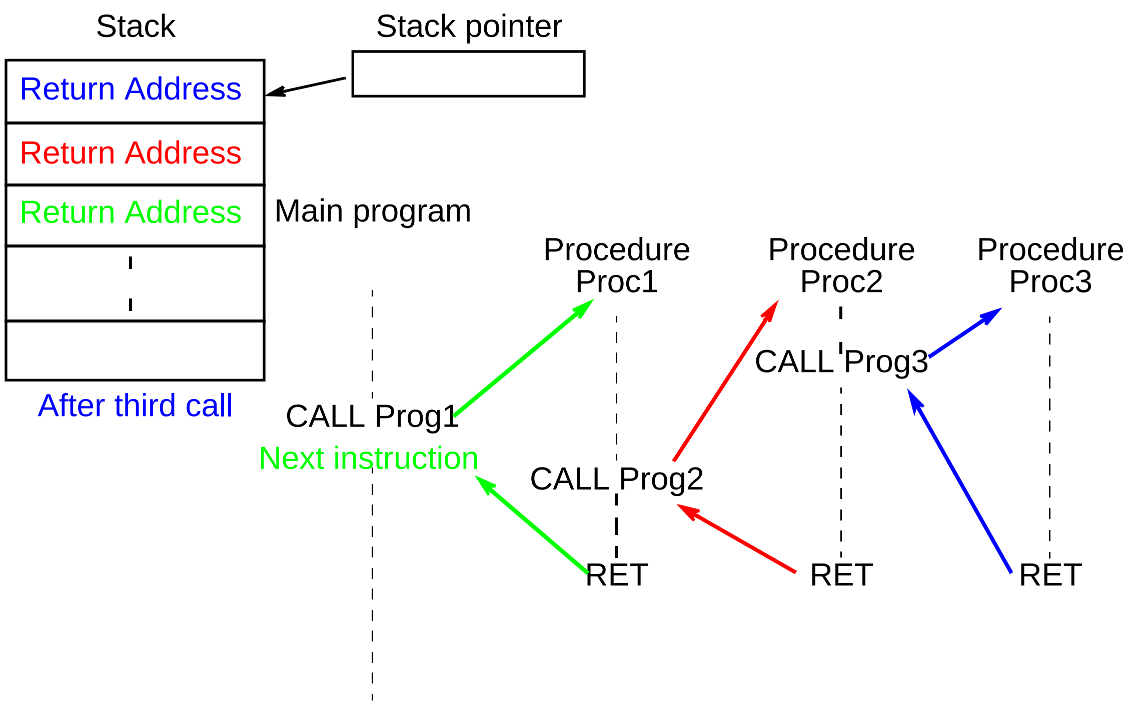
As items are placed on the stack, the stack grows, and as items are removed from the stack, the stack contracts. Although not shown in figures here, normally memory stacks are made to grow downwards, i.e., items are added to locations with decreasing addresses. Why?





Nested Procedure Calls

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



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

Part of CALL instruction is to decrement the stack pointer by 4.

Part of RET instruction is to increment stack pointer by 4.

Passing parameters using a stack

Stack can be used to hold parameters passed to called procedure and passed back from called procedure. Processors which 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 – PUSH instruction decrements the stack pointer before (or after) an item is copied onto the stack.
- POP – POP increments the stack pointer after (or before) an item is copied from the stack.

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

Saving registers

The 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 is probably best done inside called procedure.

Stack Frame

Convenient to specify the group of locations on the 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.

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

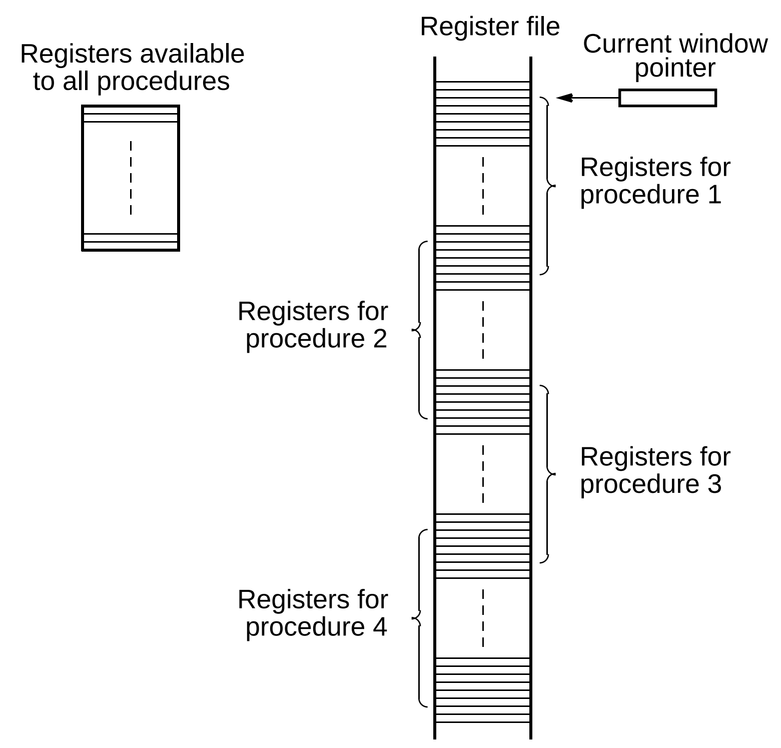
Co-routines, those routines that call each other alternately, also do not need to use a stack for the return address. A single register can be used to hold the return address. The DEC Alpha processor has a specific instruction for handling co-routines.

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.

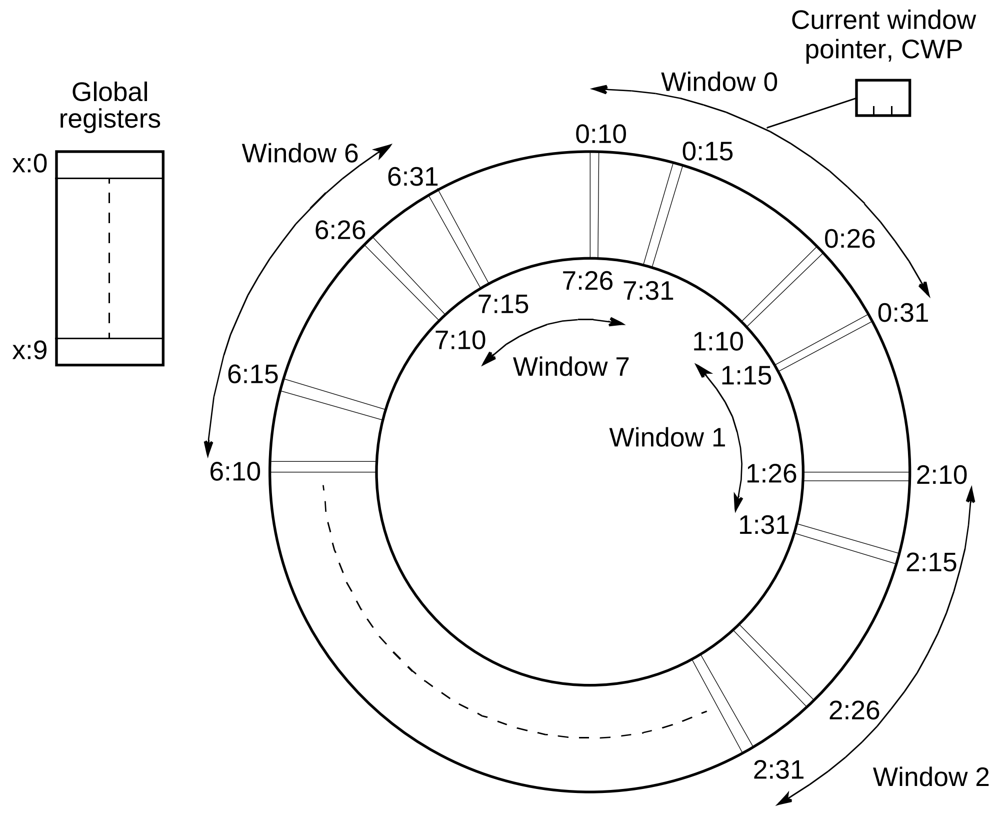
Register window

The 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 have followed Berkeley design).


Register file
Registers available to all procedures
Current window pointer
Registers for procedure 1
Registers for procedure 2
Registers for procedure 3
Registers for procedure 4

RISC register window



RISC II register window

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 R31,0[R29]  ;After return from call, restore
               ;previous return address in R31
ADD R29,R29,4  ;Increment stack pointer
```

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

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

To store parameters as well on the 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
```

Pipelined Processor Design

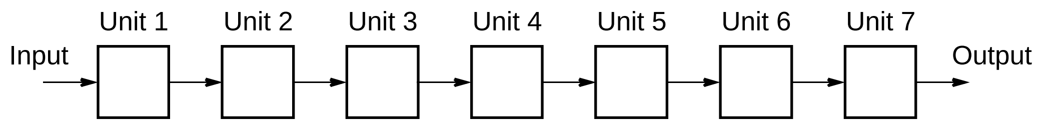
A basic techniques to improve the performance - *pipeline technique*, always applied in high performance systems. Adapted in RISCs and all processors.

The operation of the processor can be divided into a number of steps, e.g.:

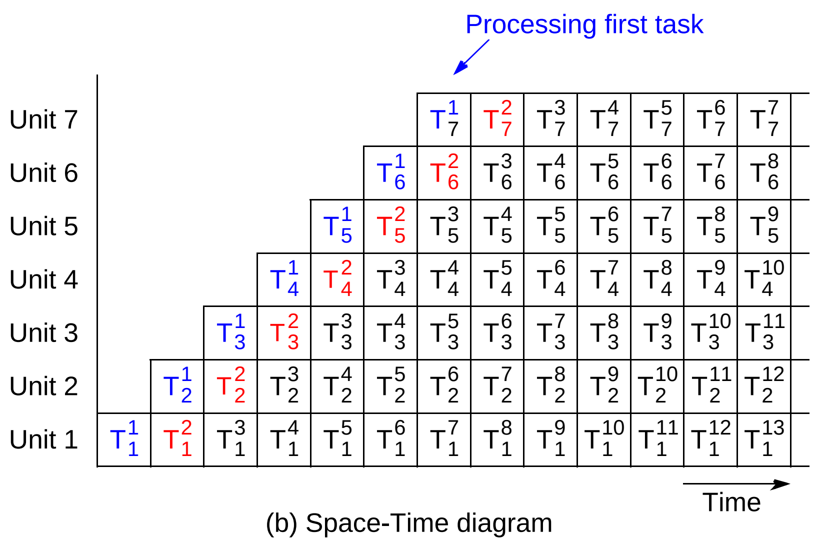
1. Fetch instruction.
2. Fetch operands.
3. Execute operation.
4. Store results

or more steps.

Space-Time Diagram



(a) Units

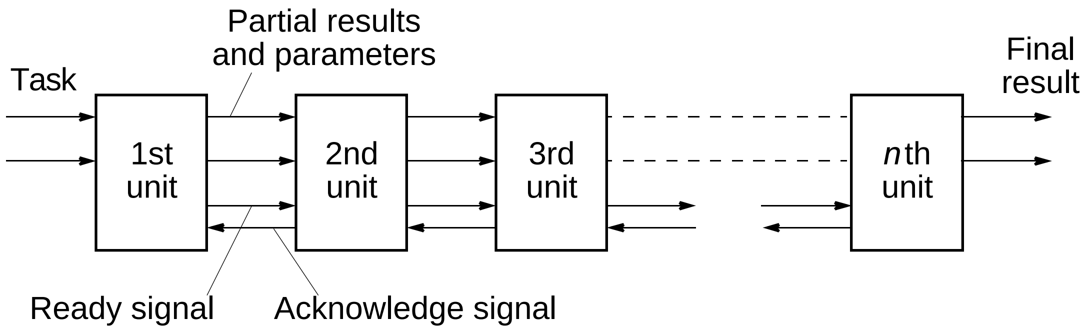

Processing first task
Unit 7: T_7^1 T_7^2 T_7^3 T_7^4 T_7^5 T_7^6 T_7^7
Unit 6: T_6^1 T_6^2 T_6^3 T_6^4 T_6^5 T_6^6 T_6^7 T_6^8
Unit 5: T_5^1 T_5^2 T_5^3 T_5^4 T_5^5 T_5^6 T_5^7 T_5^8 T_5^9
Unit 4: T_4^1 T_4^2 T_4^3 T_4^4 T_4^5 T_4^6 T_4^7 T_4^8 T_4^9 T_4^{10}
Unit 3: T_3^1 T_3^2 T_3^3 T_3^4 T_3^5 T_3^6 T_3^7 T_3^8 T_3^9 T_3^{10} T_3^{11}
Unit 2: T_2^1 T_2^2 T_2^3 T_2^4 T_2^5 T_2^6 T_2^7 T_2^8 T_2^9 T_2^{10} T_2^{11} T_2^{12}
Unit 1: T_1^1 T_1^2 T_1^3 T_1^4 T_1^5 T_1^6 T_1^7 T_1^8 T_1^9 T_1^{10} T_1^{11} T_1^{12} T_1^{13}
Time

(b) Space-Time diagram

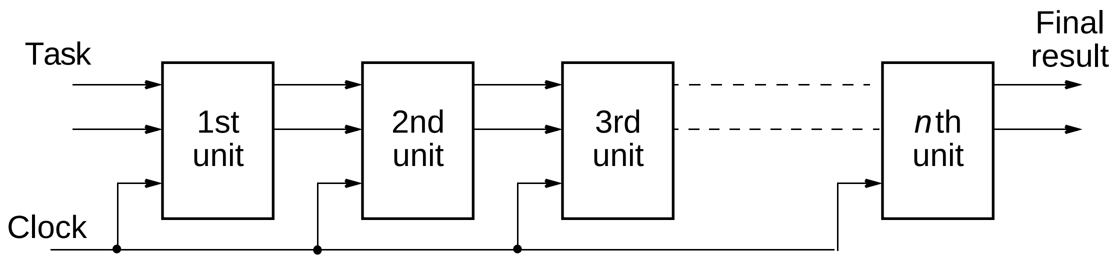
Pipeline data transfer

Two methods of implementing the information transfer in a pipeline:

1. Asynchronous method.
2. Synchronous method.



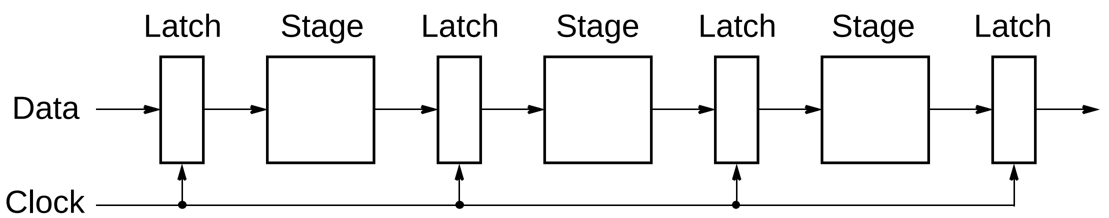
(a) Asynchronous method



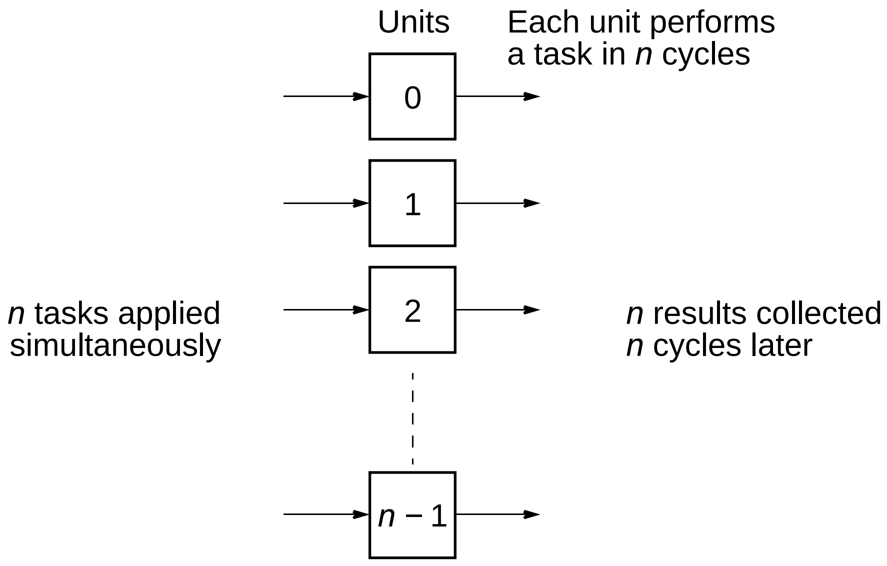
(b) Synchronous method

Synchronous method - Pipeline Staging Latches

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



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.


Units
Each unit performs a task in *n* cycles
0
1
2
n tasks applied simultaneously
n results collected *n* cycles later
n − 1

Replicated units

Speed-Up

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

$$\text{Speed-up} = \frac{T_1}{T_n} = \frac{sn}{n + (s - 1)}$$

The potential maximum speed-up is *n*, though this would only be achieved for an infinite stream of tasks and no hold-ups in the pipeline.

Efficiency

Efficiency is given by:

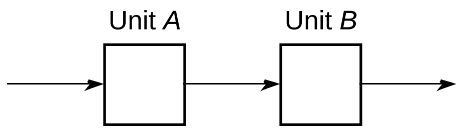
$$\text{Efficiency} = \frac{\sum_{i=1}^{n} t_i}{n \times (\text{overall operating time})}$$

$$= \frac{s}{n + (s - 1)}$$

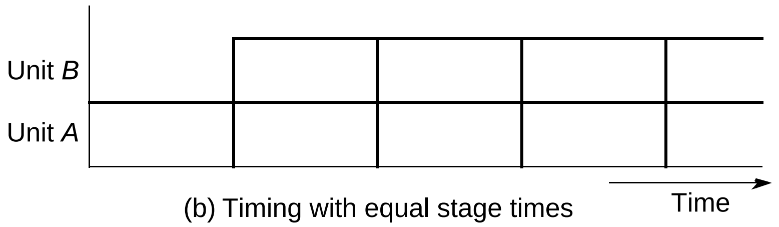
$$= \frac{\text{Speed-up}}{n}$$

where t_i is time unit *i* operates.

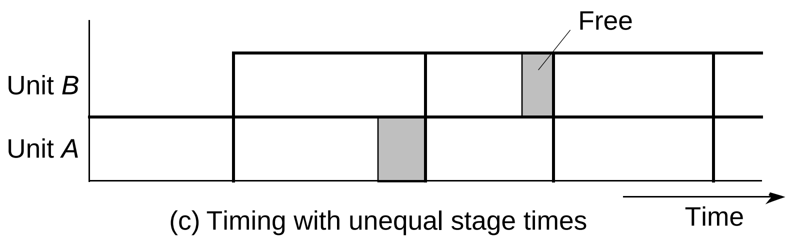
Two stage overlap



(a) Stages



(b) Timing with equal stage times



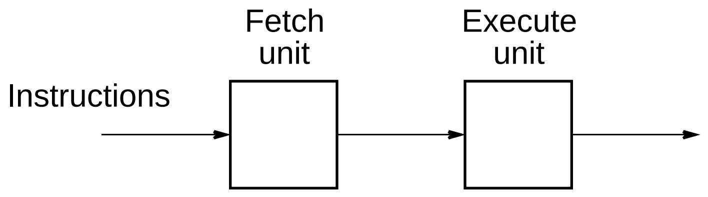
(c) Timing with unequal stage times

Overall processing time given by:

$$\text{Processing time} = \sum_{i=1}^{n-1} \text{Max}(T(A_i), T(B_{i-1}))$$

where $T(A_i)$ = time of ith operation in A, and $T(B_i)$ = time of ith operation in B.

Two Stage Fetch /Execute Pipeline



(a) Fetch/execute stages

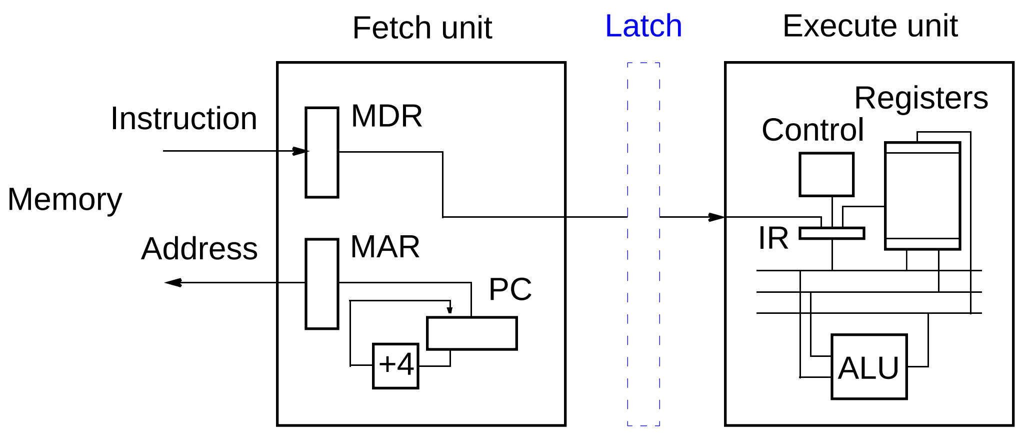
EX		Execute 1st instruction	Execute 2nd instruction	Execute 3rd instruction
IF	Fetch 1st instruction	Fetch 2nd instruction	Fetch 3rd instruction	Fetch 4th instruction

Time →

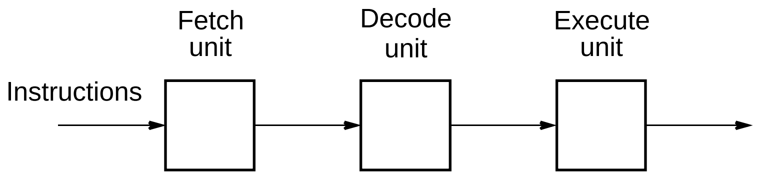
IF = Fetch unit
EX = Execute unit

(b) Space-time diagram with ideal overlap

A Two-Stage Pipeline Design



Fetch/decode/execute pipeline



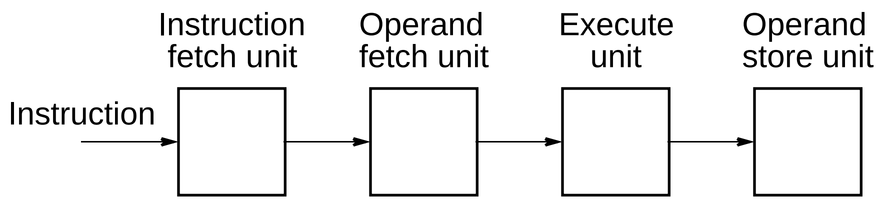
(a) Fetch/decode/execute stages

Execute			Execute 1st instruction	Execute 2nd instruction	Execute 3rd instruction
Decode		Decode 1st instruction	Decode 2nd instruction	Decode 3rd instruction	Decode 4th instruction
Fetch	Fetch 1st instruction	Fetch 2nd instruction	Fetch 3rd instruction	Fetch 4th instructions	Fetch 5th instruction

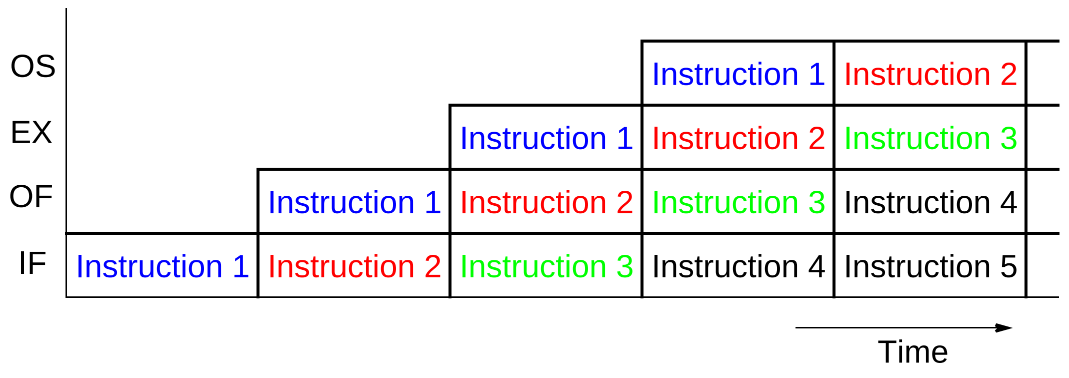
Time $\longrightarrow$

(b) Ideal overlap

Four-Stage Pipeline



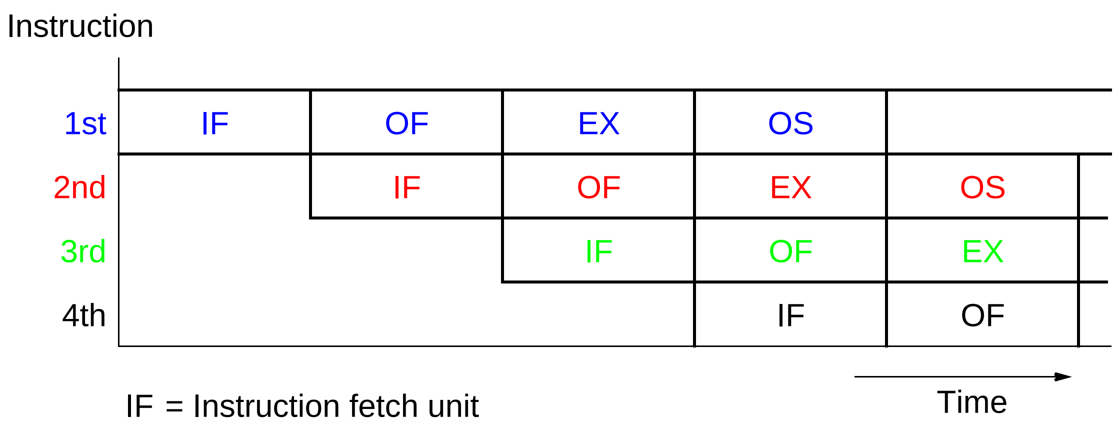
Four-Stage Pipeline Space-Time Diagram



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

Four-stage Pipeline “Instruction-Time Diagram”

An alternative diagram:



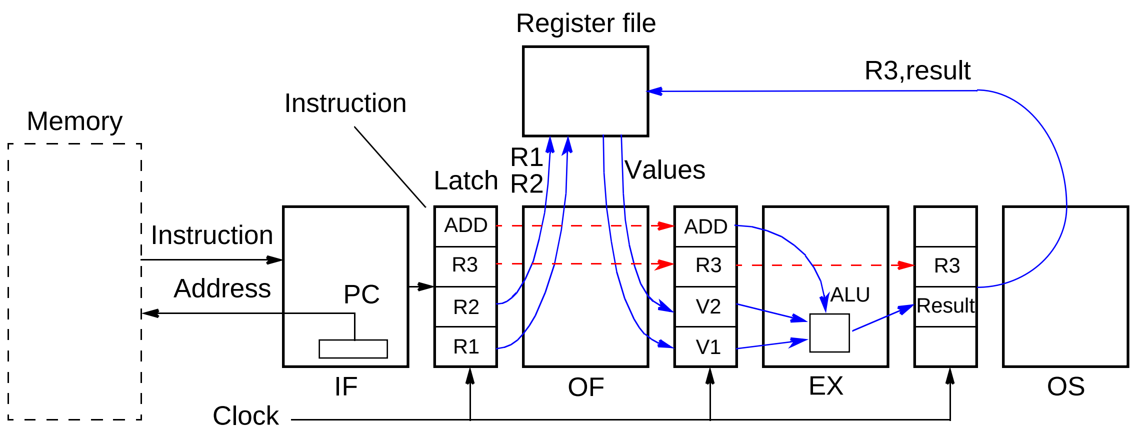
Instruction					
1st	IF	OF	EX	OS	
2nd		IF	OF	EX	OS
3rd			IF	OF	EX
4th				IF	OF

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

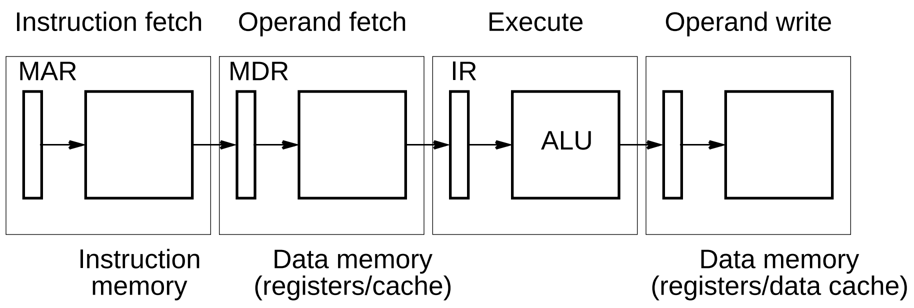
Information Transfer in Four-Stage Pipeline

Register-Register Instructions

ADD R1, R2, R3



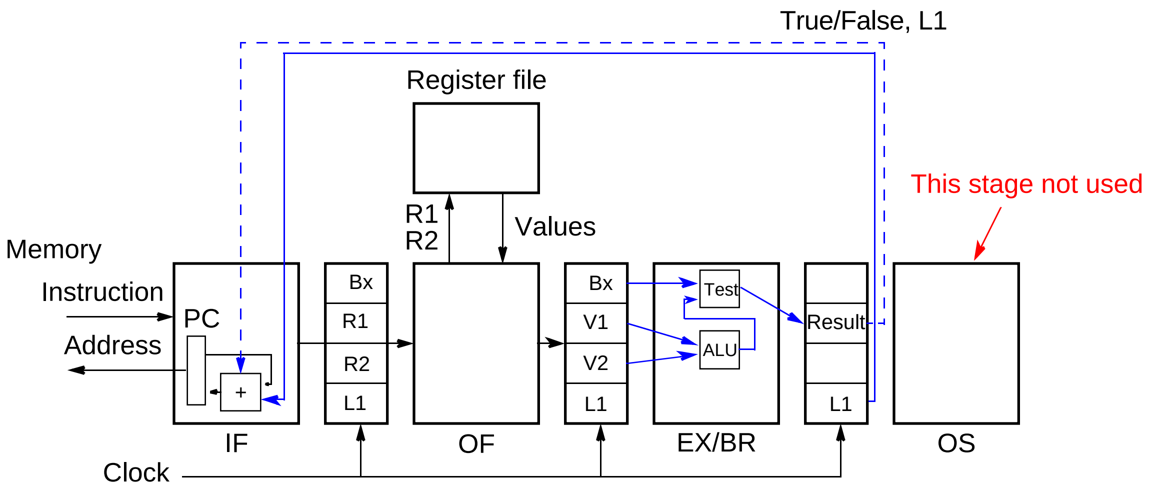
Alternative way of depicting pipeline showing data register twice



Four-Stage Pipeline

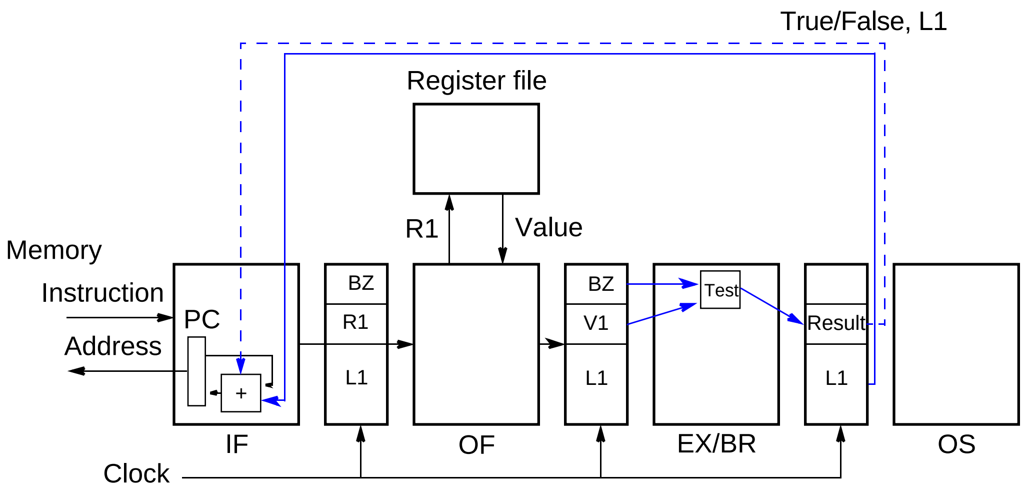
Branch Instructions

Bcond R1, R2, L1



Simpler Branch Instruction

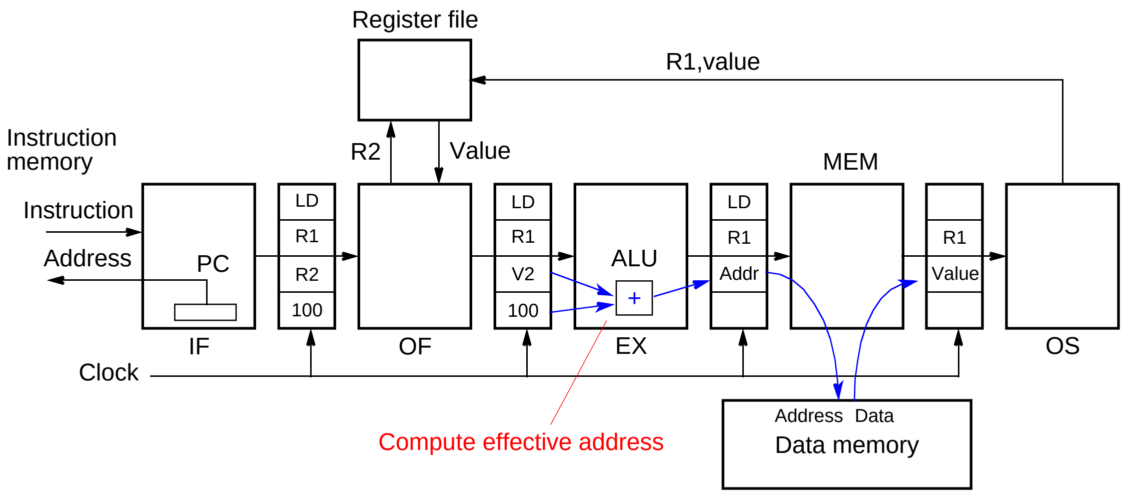
BZ R1, L1



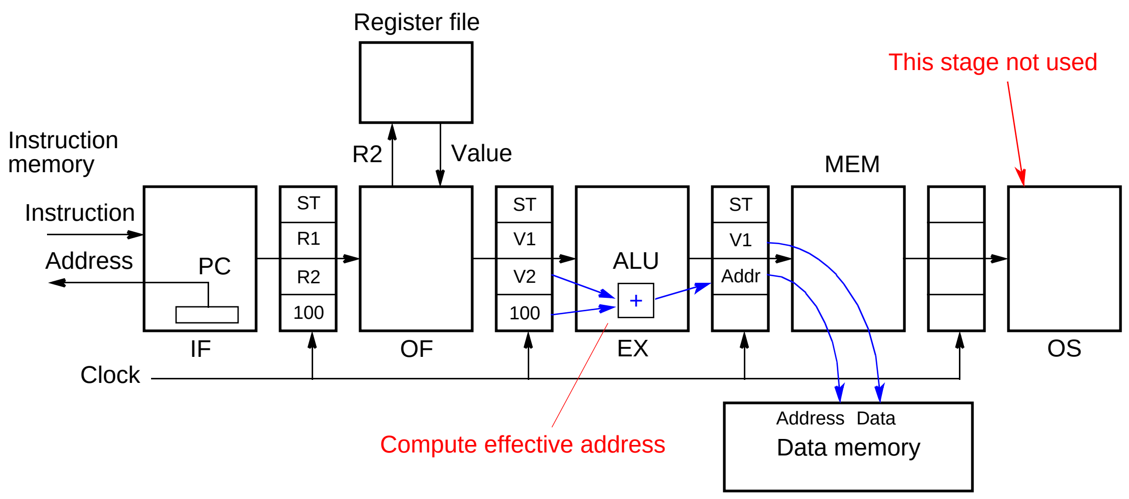
Load and Store Instructions

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

LD R1, 100[R2]

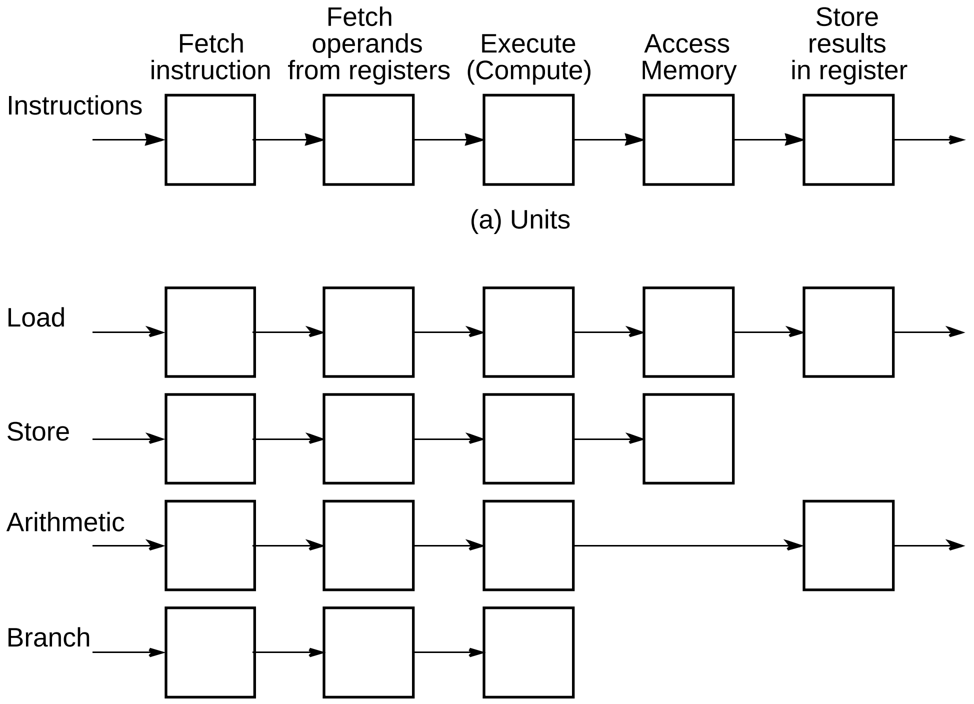


ST 100[R2], R1



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

Usage of Stages



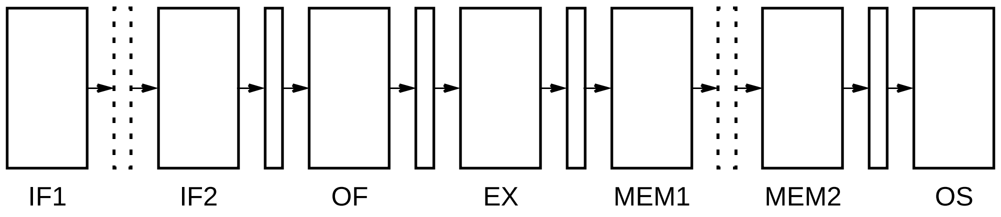
(a) Units

(b) Instruction usage

More Pipeline Stages

Example

Seven-stage instruction pipeline with two stages for accessing memory



Instruction Pipeline Hazards

Major causes for breakdown or hesitation of an instruction pipeline:

1. Resource conflicts.
2. Procedural dependencies (caused notably by branch instructions).
3. Data dependencies between instructions

All are caused by having more than one instruction being executed in the pipeline at any instant.

Resource conflicts

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

Solutions

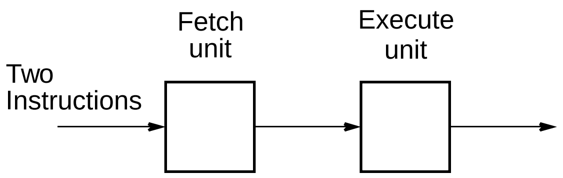
Resource conflicts can always be resolved by duplicating the resource, for example having two memory module ports or two functional units.

For main memory/cache memory conflicts

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

Early solution

Fetch two instructions and then execute them sequentially.



(a) Fetch/execute stages

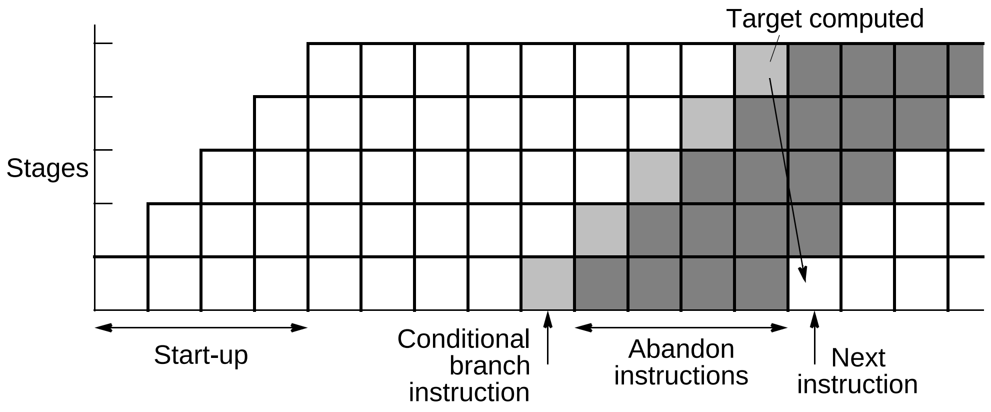
Decode		Execute 1st instruction	Execute 2nd instruction	Free	Execute 3rd instruction
Fetch	Fetch 1st/2nd instructions	Free	Free	Fetch 3rd/4th instructions	Free

Time →

(b) Fetching two instructions simultaneously

Fetch/execute overlap

Procedural dependencies and branch instructions



Effect of conditional branch instruction in a pipeline

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

Example of effect

- Five-stage pipeline
- 10 ns steps
- Instruction which subsequently cleared the pipeline at the end of its execution occurred every ten 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.

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).

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: BL  R5,R4,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`**.

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

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

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

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

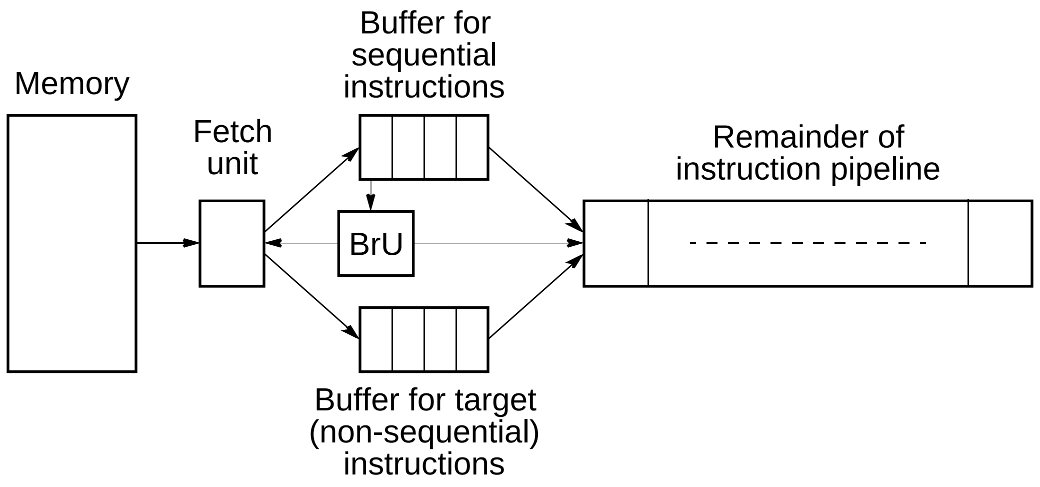
```
L2:      .
       Loop body
         .
     BEQ R4,R5,L2
```

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

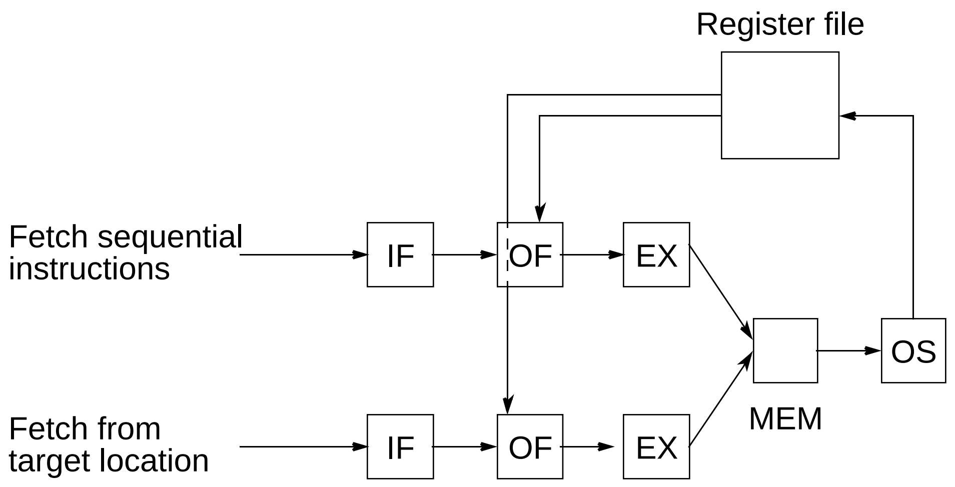
Strategies to reduce number of times pipeline breaks down due to conditional branch instructions

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.

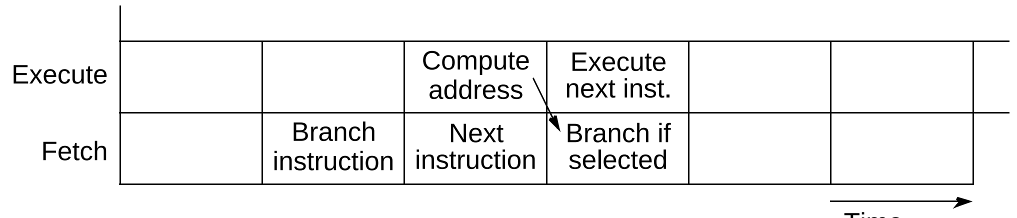
Instruction buffers


Memory
Fetch unit
Buffer for sequential instructions
BrU
Remainder of instruction pipeline
Buffer for target (non-sequential) instructions

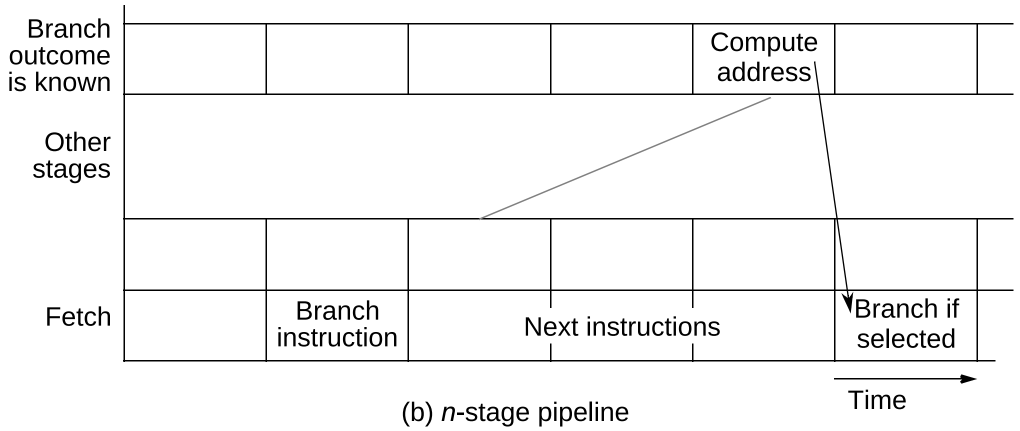
Replicating first stages of pipeline



Delayed branch instructions



(a) Two-stage pipeline



(b) *n*-stage pipeline

Example

```
        ADD R3,R4,R5
        SUB R2,R2,1
        BEQ R2,R0,L1
            .
    L1:     .
            .
```

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

```
        SUB  R2,R2,1
        BEQD R2,R0,L1
        ADD  R3,R4,R5
            .
    L1:     .
            .
```

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

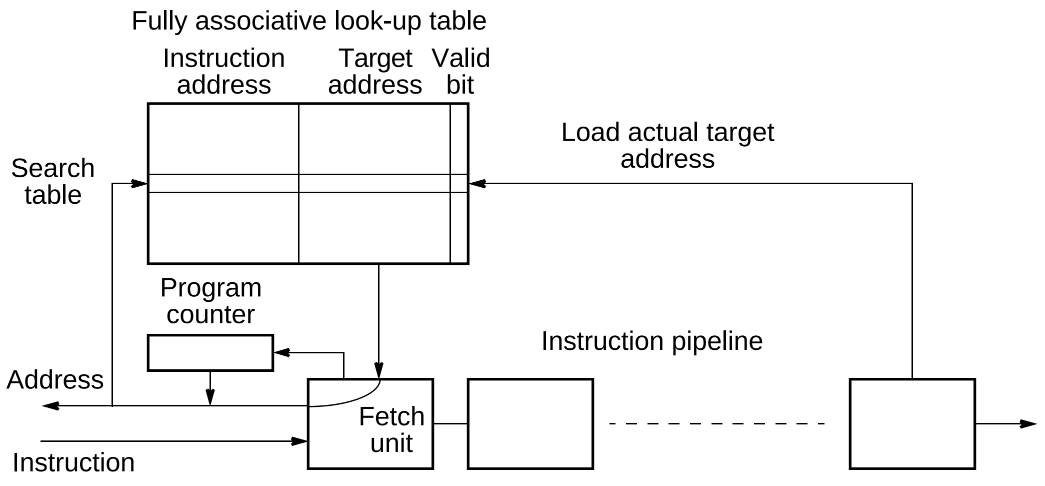
```
        SUB R2,R2,1
        BEQ R2,R0,L1
        NOP
            .
            .
    L1:     .
            .
```

A compiler can easily insert these NOPs.

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 history table (prediction logic not shown – sequential instructions taken until correct target address loaded)

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.

Two-bit prediction

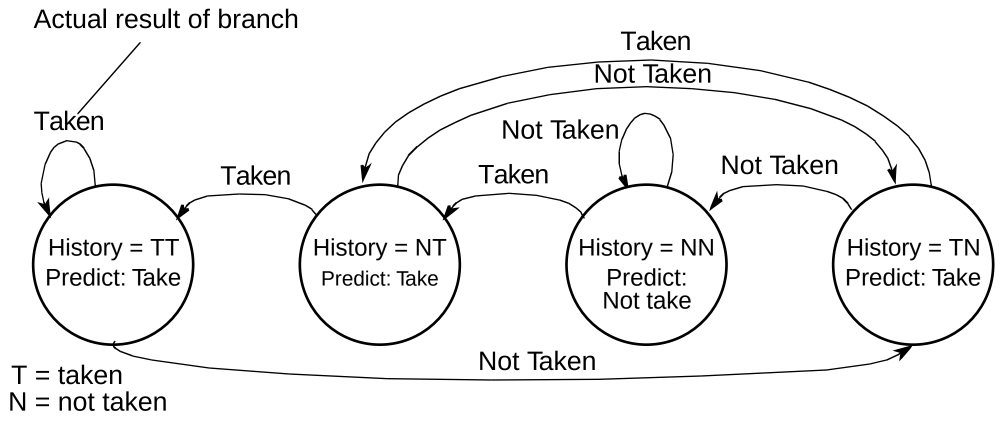
One two-bit prediction algorithm is based upon the history of *previous actions of a branch instruction* as shown below.

Two-bit prediction

History of branch actions		Prediction
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

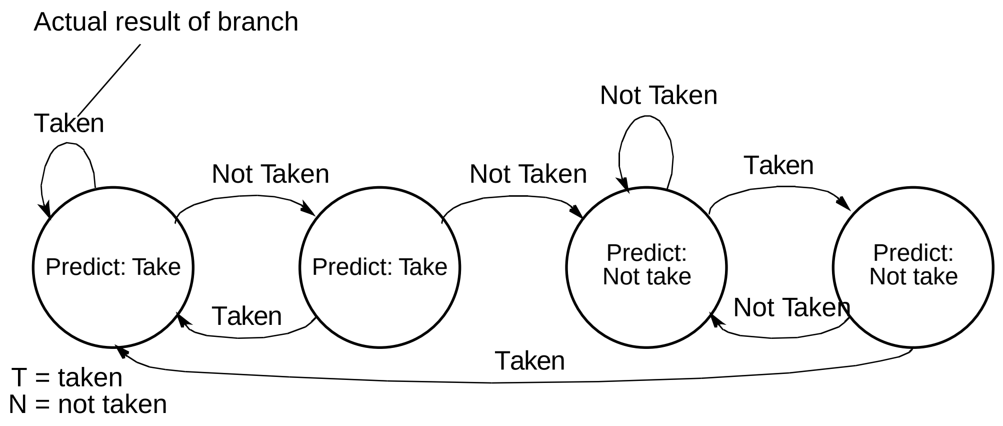


Based upon the history of the branch *predictions*, that is, the prediction is only changed after two mispredictions, as described below.

Alternative two-bit prediction

History of predictions		Prediction
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

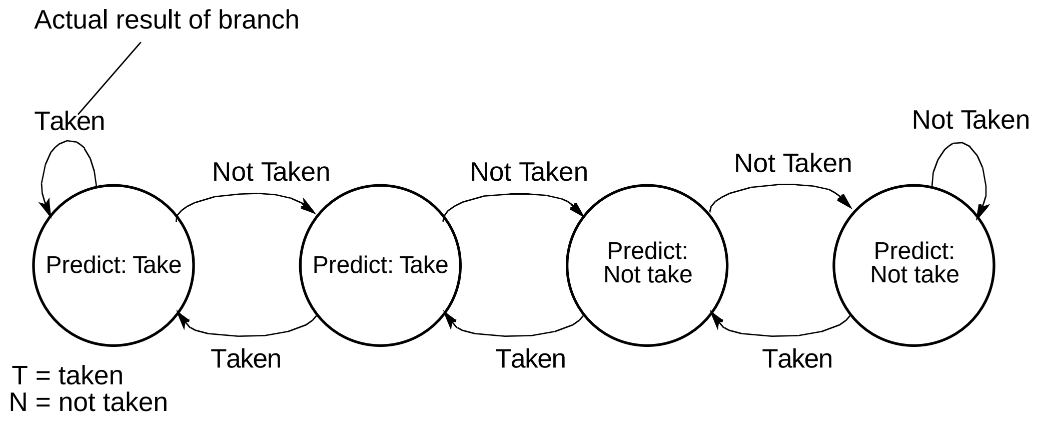
Two-bit predictor based upon history of predictions



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, or different predictors could be used.)

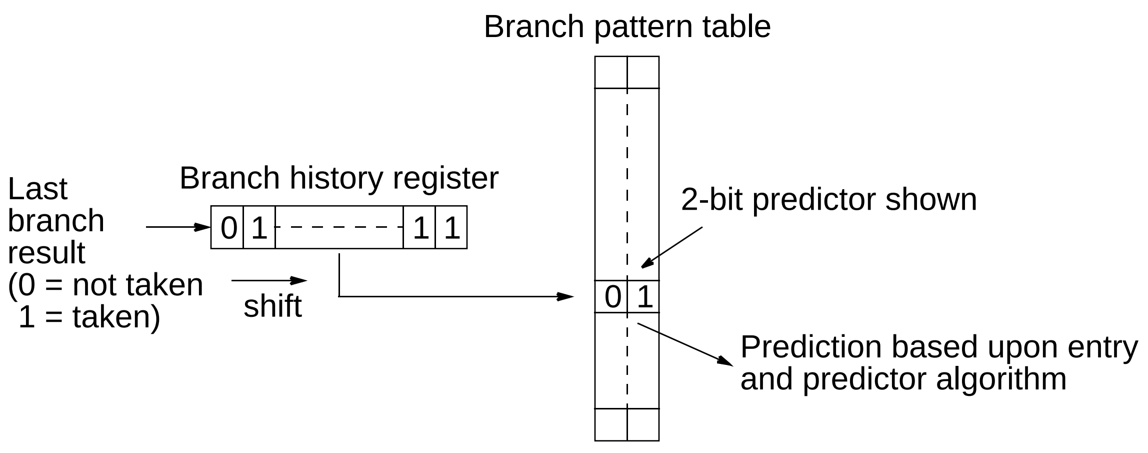


Two-bit saturation counter predictor

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 instruction generally is recorded.


Branch pattern table
Last branch result (0 = not taken 1 = taken)
Branch history register
0 1 1 1
shift
2-bit predictor shown
0 1
Prediction based upon entry and predictor algorithm

Two-level (adaptive) predictor

Static prediction

Static prediction makes the prediction before execution.

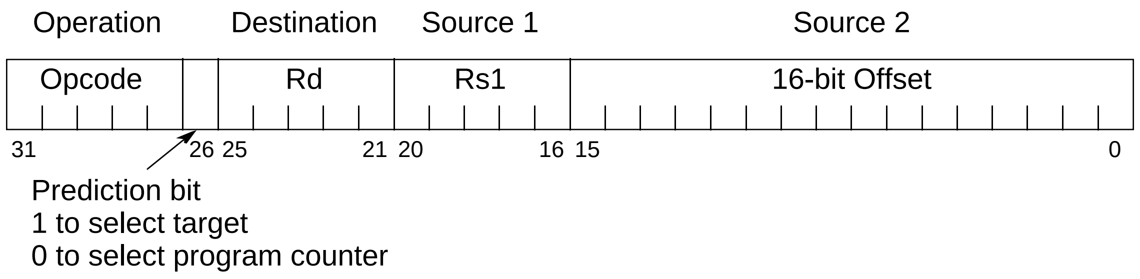
A very simple hardware prediction – always chooses one way (either always taken, or always not taken).

"Predict never taken" is done in the Motorola 68020 - essentially no prediction.

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

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.

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.

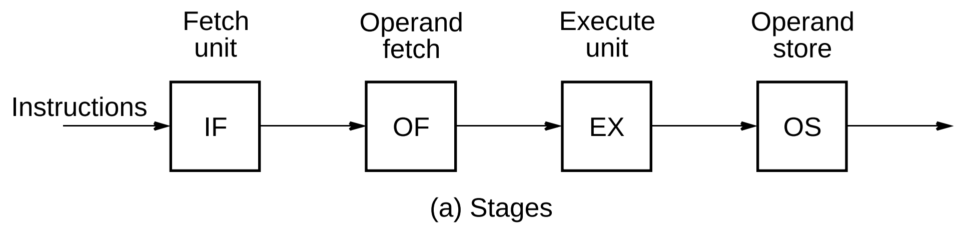
Example

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

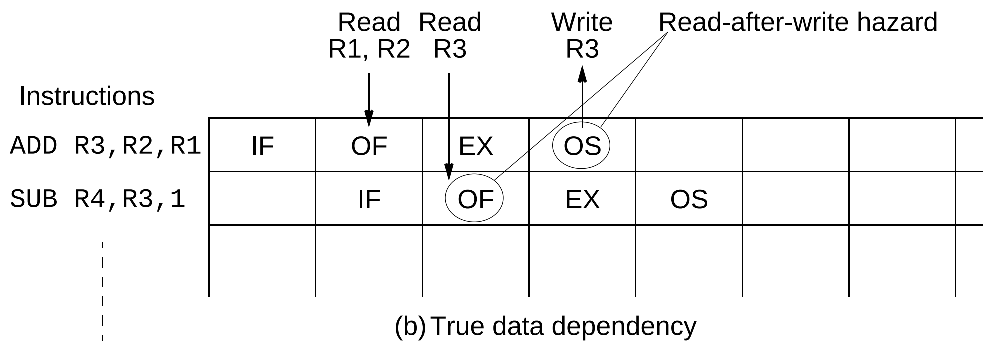
“data” dependency between instruction 1 and 2.

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,R3,1    ;R2 = R3 - 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 R2.

Output dependencies in a single pipelne

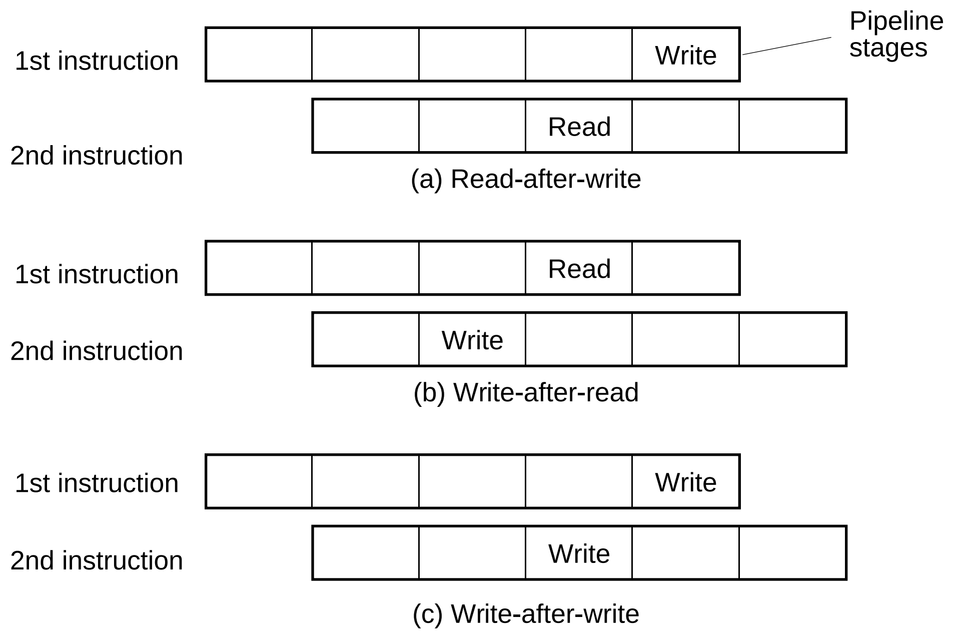
Again the dependency would not be significant if all instructions write at the same time in the pipeline and instructions are processed in program order. Output dependencies are a form of resource conflict, because the register in question, R3, is accessed by two instructions. The register is being reused.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

Detecting hazards

Can be detected by considering read and write operations on specific locations accessible by the instructions. In terms of two operations, *read* and *write*, operating upon a single location:

- 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.



Read/write hazards

Matematical Conditions for Hazard
(Berstein's Conditions)

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

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

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

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) ∩ (R1) = φ, (R2,R1) ∩ (R5) = φ, (R3) ∩ (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.

Pipeline interlocks

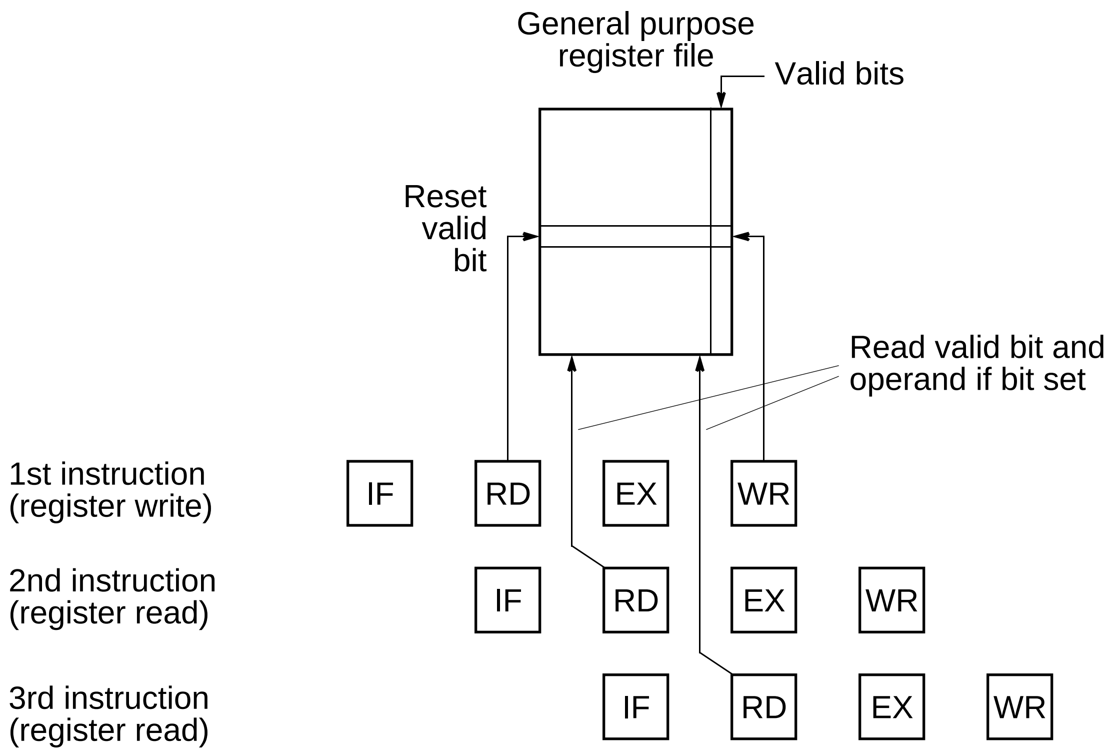
A relatively simple method of maintaining a proper sequence of read/write operations is to associate a 1-bit tag with each operand register.

This tag indicates whether a valid result exists in the register, say 0 for not valid and 1 for valid.

A fetched instruction which will write to the register examines the tag and if the tag is 1, it sets the tag to 0 to show that the value will be changed.

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

Any instruction fetched before the operand tags have been set has to wait.


General purpose register file
Valid bits
Reset valid bit
Read valid bit and operand if bit set
1st instruction (register write)
2nd instruction (register read)
3rd instruction (register read)
IF
RD
EX
WR

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

Example

Suppose the instruction sequence is:

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

There is a read-after write dependency between instruction 1 and instruction 2 (R3) and a read-after-write dependency between instruction 1 and instruction 3 (again R3).

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

Both instructions 2 and 3 must examine the valid bit of their source registers prior to reading the contents of the 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 instruction 1 and instruction 2. Instruction 1 will reset valid bit of R3 in preparation to altering its value, and move through the pipeline. Instruction 2 immediately behind it would also reset valid bit of R3 because it too will alter its value, but will find the valid bit already reset. If instruction 2 were to be allowed to continue, instruction 3 immediately behind instruction 2 would only wait for the valid bit to be set, which would first occur when instruction 1 writes to R3. Instruction 3 would get value generated by instruction 1, rather than value generated by instruction 2 as called for in the program sequence.

Correct algorithm for setting valid bit

```
WHILE destination register valid bit = 0 wait.
Set destination register valid bit to 0 and proceed.
```

Forwarding

Refers to technique of passing result of one instruction directly to another instruction to eliminate the 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 are normally faster than accesses to memory locations.

Forwarding can also be applied at the hardware level to eliminate pipeline cycles for reading registers that were updated in a previous pipeline stage. In that case, eliminates register accesses by using faster data paths.

Internal forwarding

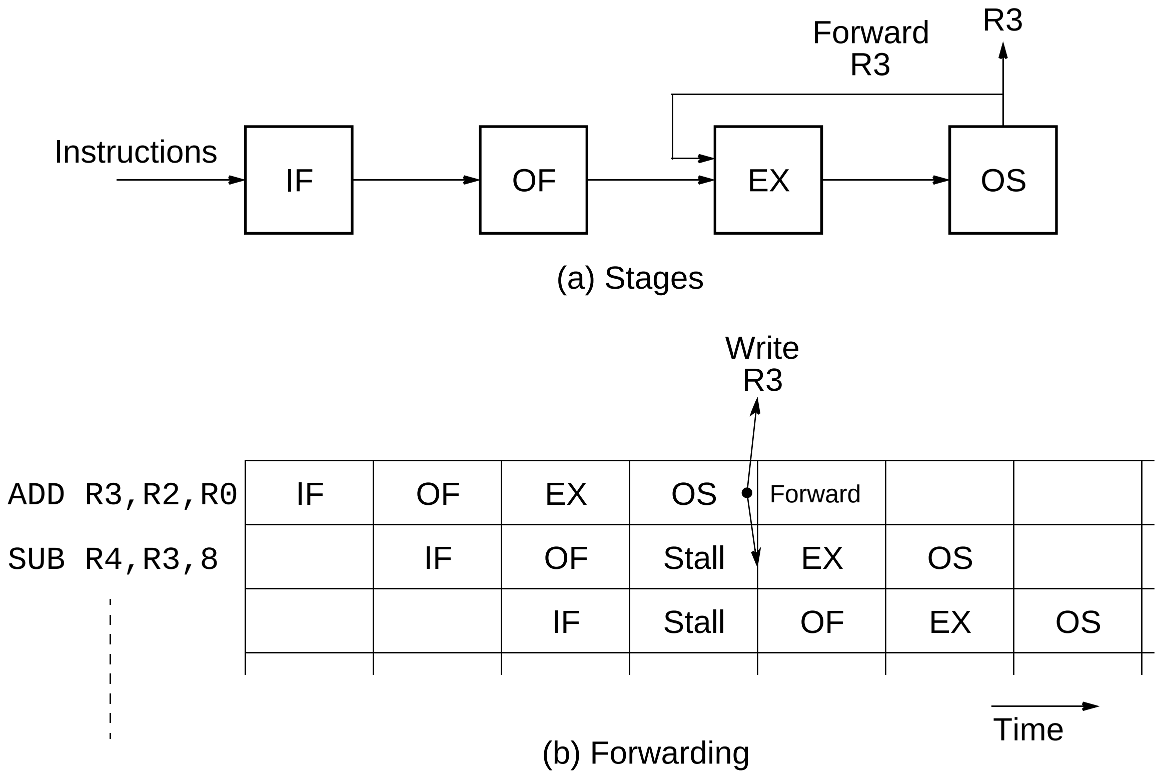
Internal forwarding is hardware forwarding implemented by processor registers or data paths not visible to the programmer.

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.

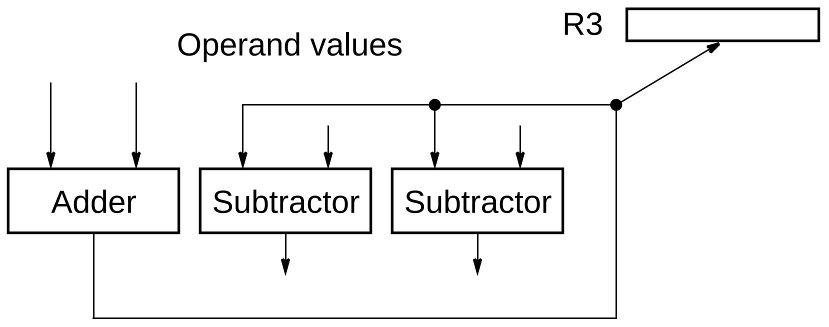
Internal forwarding forwards the value being stored in R3 by the operand store unit directly to the execute unit.



Internal forwarding in a four stage pipeline

The 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.



Forwarding using multiple functional units

Higher performance processor design

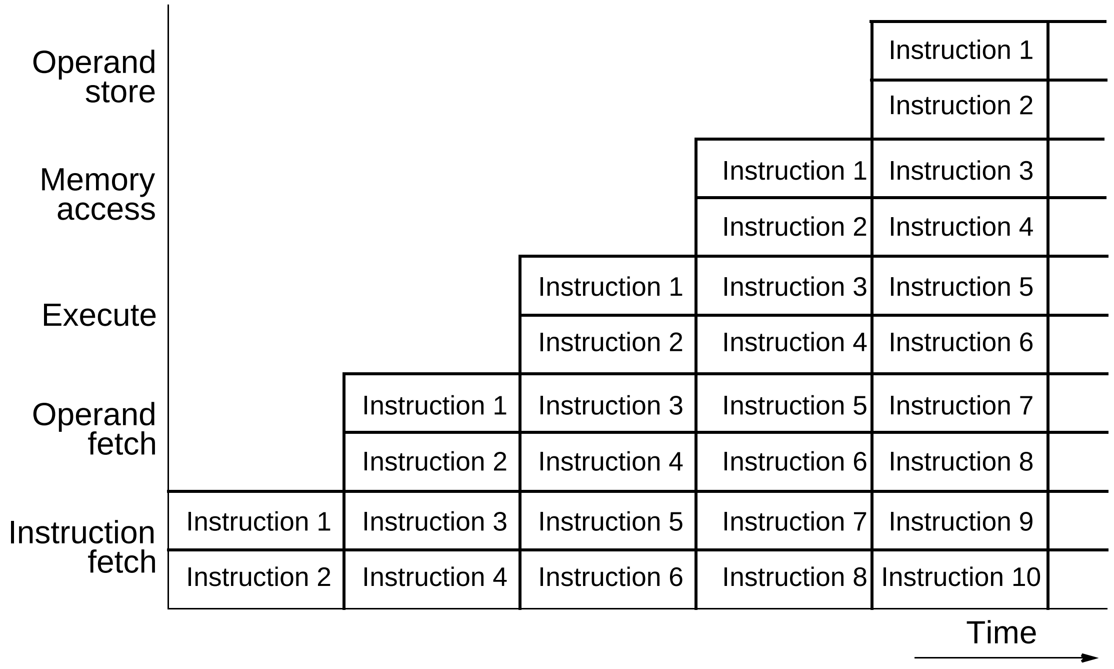
Superscalar processors

A conventional “scalar” processor executes scalar instructions, i.e., instructions operating upon single operands such as integers.

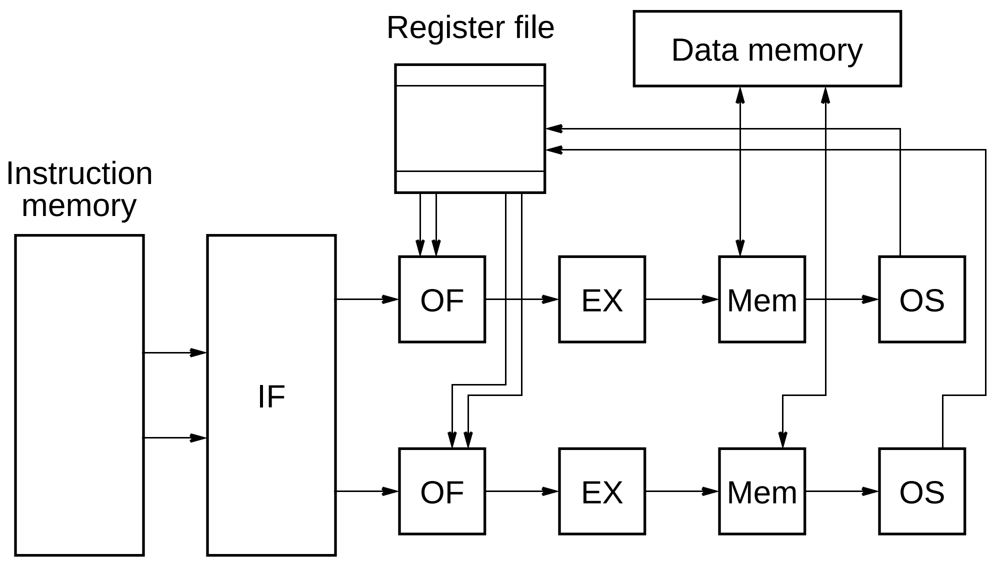
A *superscalar* processor is a processor which executes more than one (scalar) instruction concurrently.

Superscalar operation is achieved by fetching more than one instruction simultaneously, and then executing more than one instruction simultaneously.

Given a pipeline structure, a superscalar processor requires more than one pipeline, one pipeline for each instruction to be processed concurrently.

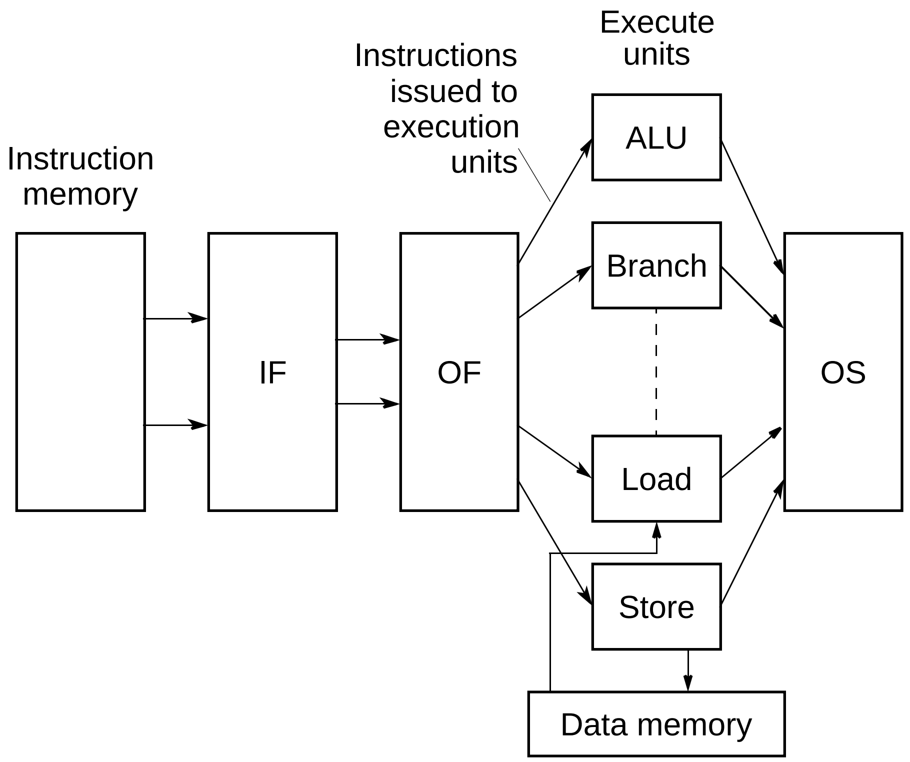


Superscalar processor timing with two pipelines



Dual pipeline processor

Example - Original Pentium processor



Superscalar design with specialized execution units

Example - Pentium Pro/II processors

In-order issue of instructions – Instructions send for execution in program order

Out-of-order issue of instructions – Instructions send for execution not in program order (instructions allowed to overtake stalled instructions).

Either way, usually instructions may finish execution out-of-order (*out-of-order completion*).

In-order completion is rarely enforced. For example in the sequence:

```
MUL R1,R2,R3
ADD R4,R5,R6
```

even if we issue the MUL instruction before the ADD instruction, the MUL instruction is likely to require more cycles and will complete after the ADD instruction.
All dependencies are a definite problem in superscalar processors with their multiple pipelines and out-of-order issue/out-of-order completion.

Another factor which does not occur in scalar designs but appears in superscalar designs is a resource conflict for a functional unit.

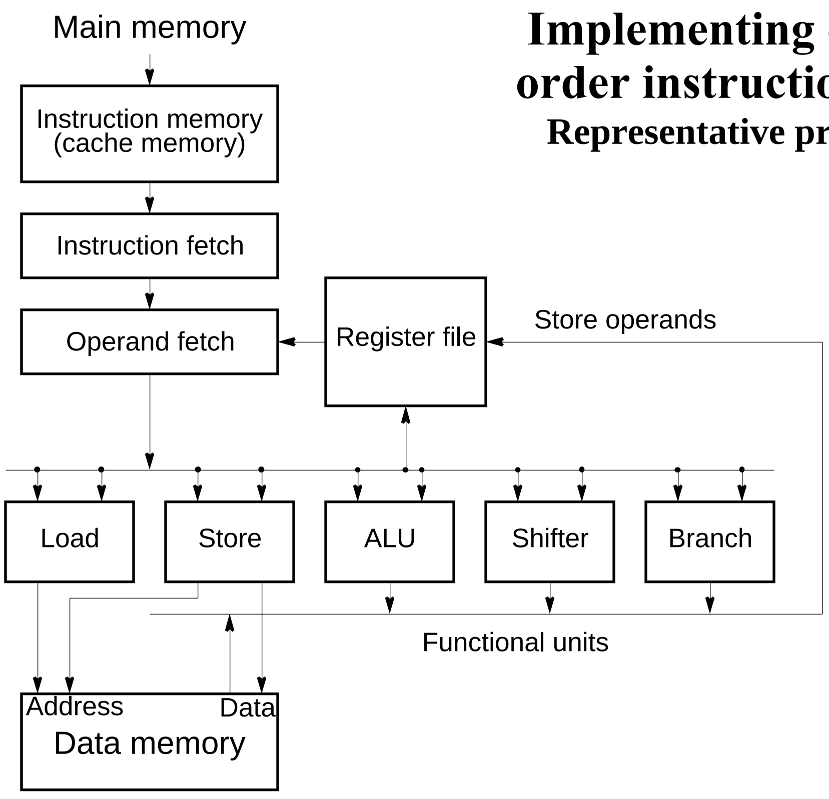
Example

```
ADD R1,R2,R3
SUB R4,R5,R6
```

No instruction dependencies. However suppose only one ALU is provided, responsible for both addition and subtraction. Clearly both ADD and SUB instructions cannot be executed together in such a design. The number of functional units provided will be a compromise between cost and possible resource conflicts.

Implementing out-of-order instruction issue

Representative processor



Superscalar (integer) processor model

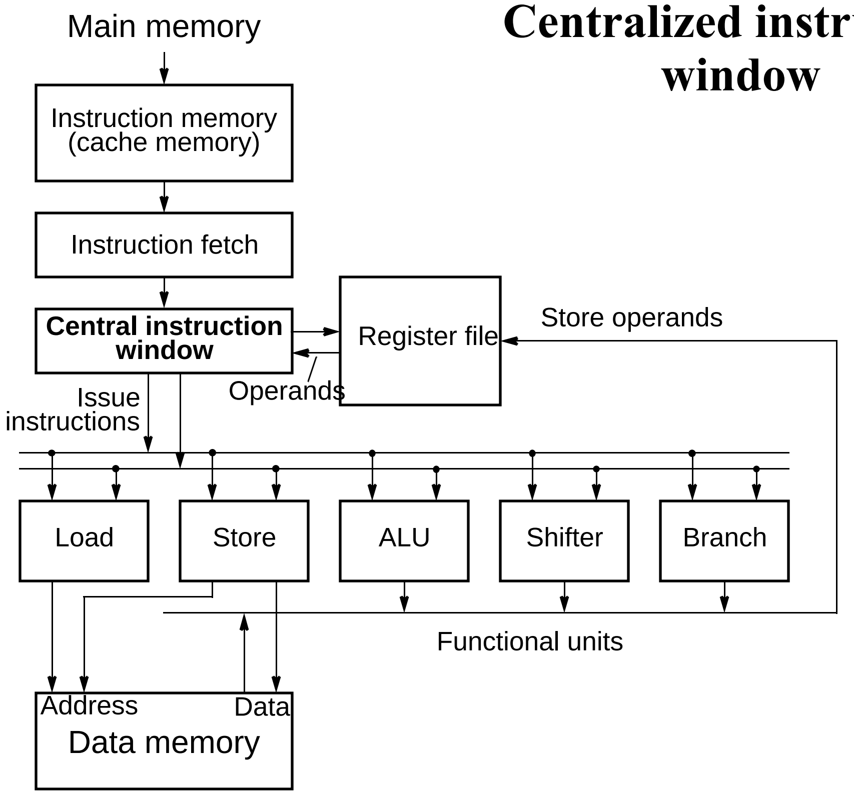
Instruction Window

To achieve out-of-order issue, an instruction buffer called an *instruction window* is used. Placed between fetch and execute stages to hold instructions waiting to be executed. Instructions are issued from the window whenever it is possible to execute the instructions, which occurs when the operands the instruction needs are available and the functional unit required for the operation is free.

The instruction window can be implemented in two ways:

1. Centralized or
2. Distributed.

Centralized instruction window



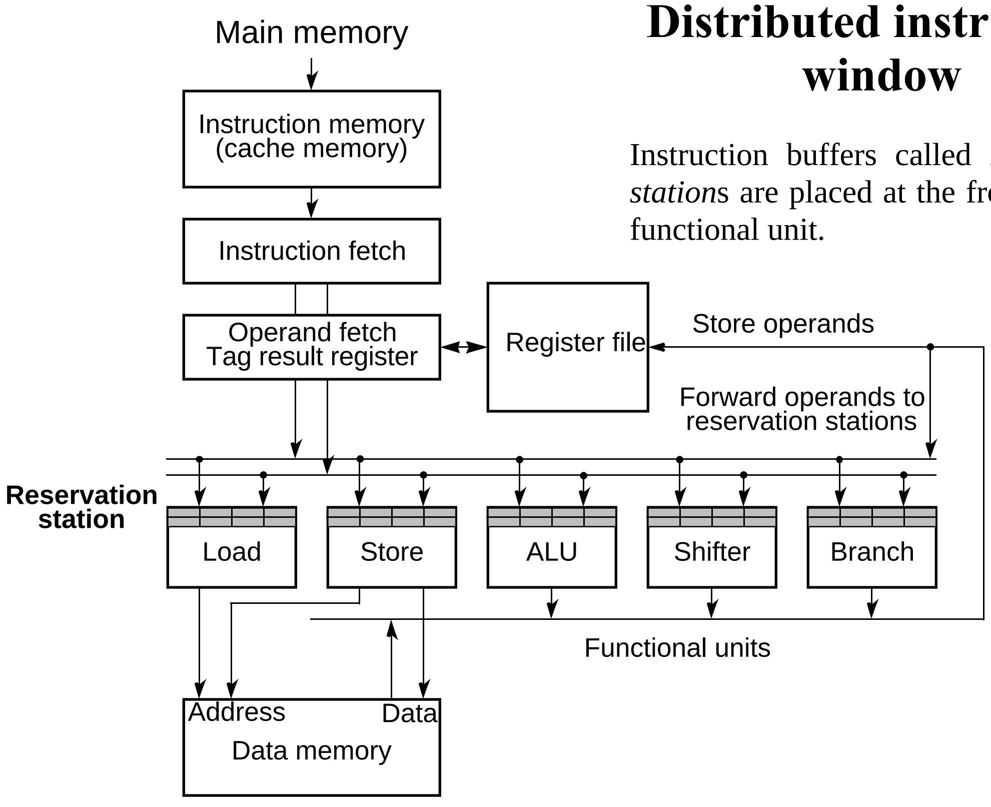
Superscalar processor with centralized instruction window

Instruction	Opcode	Destination register	Operand 1	Operand 1 register	Operand 2	Operand 2 register
1	Operation	ID	Operand value			ID
2	Operation	ID	Operand value			ID
3	Operation	ID		ID	Operand value	ID
4	Operation	ID	Operand value		Operand value	
5	Operation	ID		ID	Operand value	ID

Instruction window contents

Distributed instruction window

Instruction buffers called *reservation stations* are placed at the front of each functional unit.



Reservation stations (without renaming, see later)

Register renaming

Antidependencies and output dependencies caused essentially by reusing storage locations, i.e., they are resource conflicts. Consequently the effects of these dependencies can be reduced by duplicating the storage locations.

Example

```
ADD R1,R2,R4;R1 = R2 + R4
ADD R2,R1,1 ;R2 = R1 + 1
ADD R1,R4,R5;R1 = R4 + R5
```

has several dependencies. (It also has a resource conflict if there is only one adder.) By introducing new registers, R1* and R2*, we have:

```
ADD R1,R2,R4;R1 = R2 + R4
ADD R2*,R1,1;R2* = R1 + 1
ADD R1*,R4,R5;R1* = R4 + R5
```

eliminating the antidependency and output dependency.

Clearly we cannot keep creating new registers throughout the program.

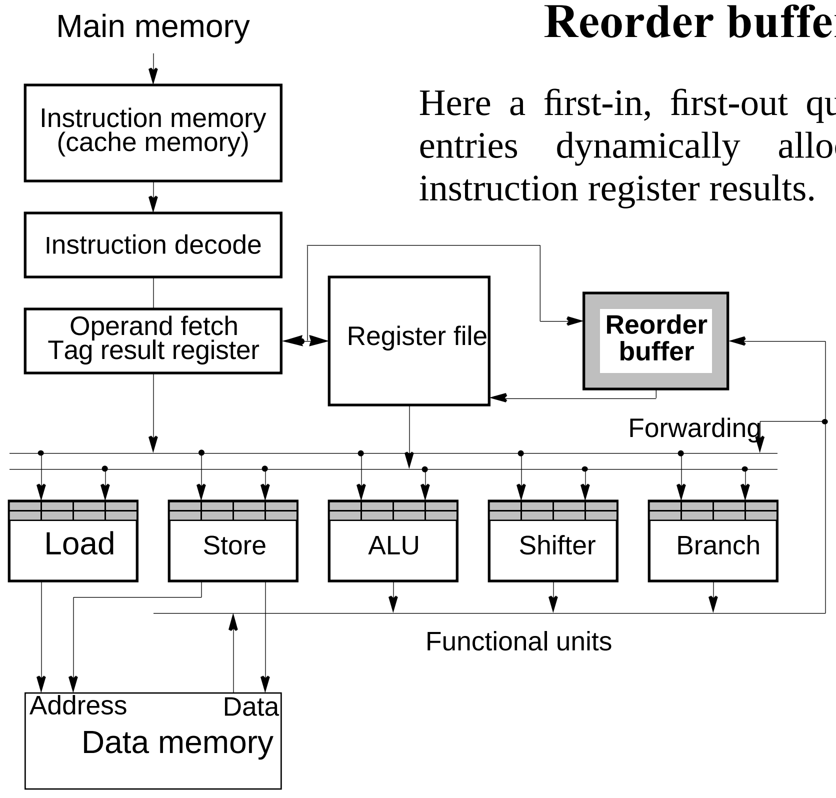
Solution is to rename existing registers temporarily. R1 might be temporarily be called R1a, R1b, R1c … as R1 is being reused in the program, and similarly for other registers, i.e.

```
ADD R1a,R2a,R4a
ADD R2b,R1a,1
ADD R1b,R4,R5
```

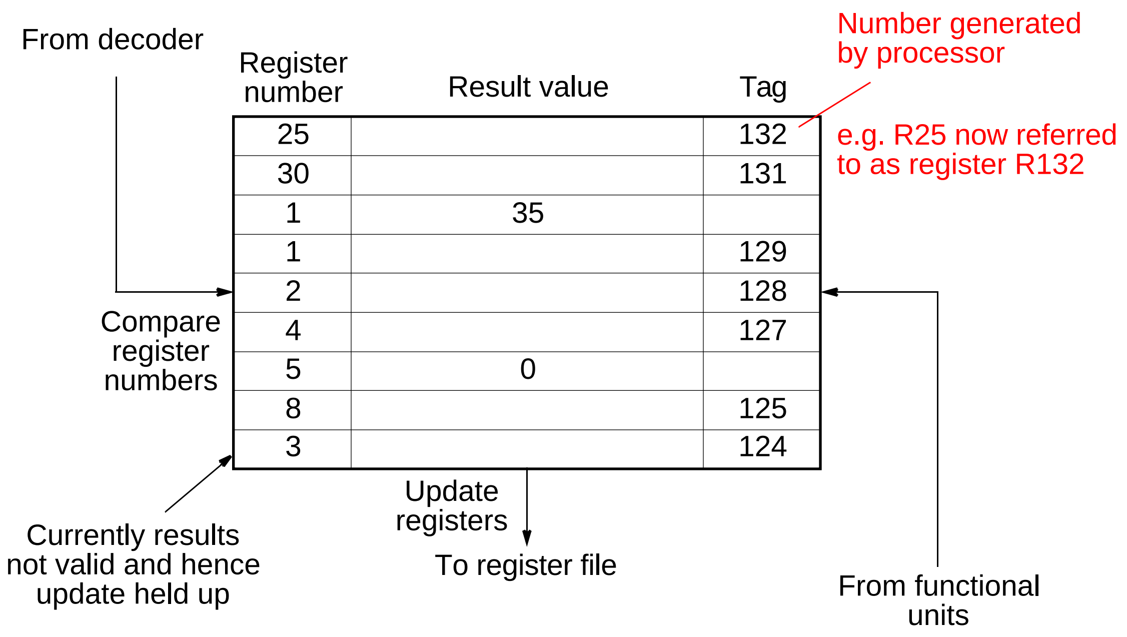
Known as *register renaming*. Can remove both antidependencies and output dependencies, and is implemented in hardware. New register instances are created and destroyed when there are no outstanding references to the stored values.

Reorder buffer

Here a first-in, first-out queue with entries dynamically allocated to instruction register results.



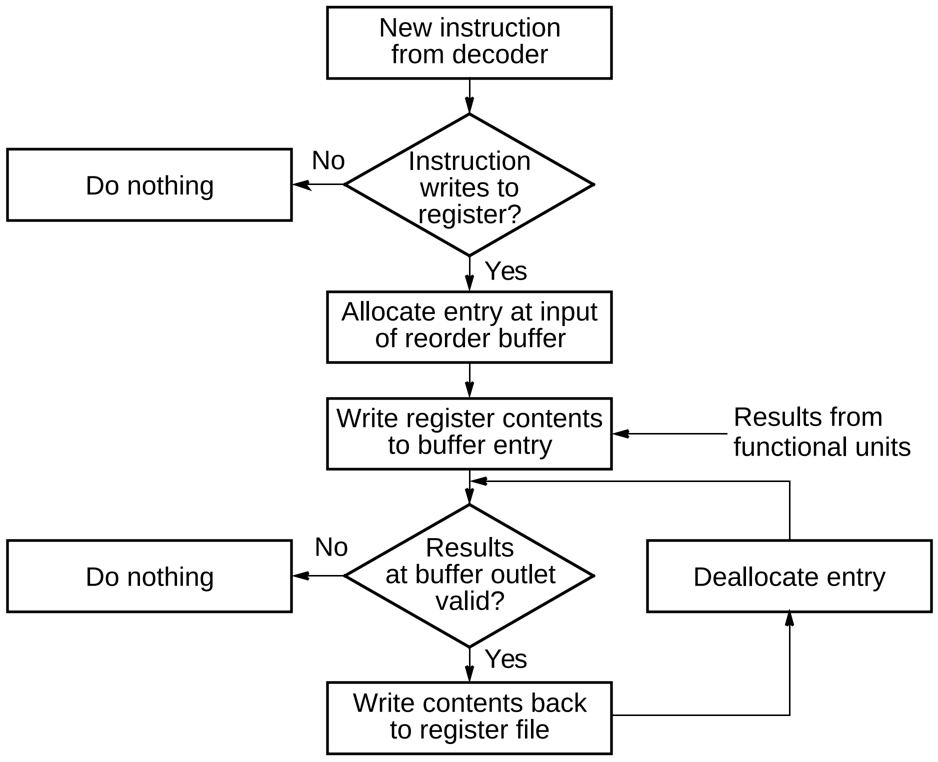
Reorder buffer with reservation stations



From decoder
Number generated by processor
e.g. R25 now referred to as register R132
Compare register numbers
Currently results not valid and hence update held up
Update registers
To register file
From functional units

Register number	Result value	Tag
25		132
30		131
1	35	
1		129
2		128
4		127
5	0	
8		125
3		124

Reorder buffer organization


New instruction from decoder
Instruction writes to register?
No
Do nothing
Yes
Allocate entry at input of reorder buffer
Write register contents to buffer entry
Results from functional units
Results at buffer outlet valid?
No
Do nothing
Deallocate entry
Yes
Write contents back to register file

Reorder buffer update algorithm

The simple valid bit approach described earlier for the single pipeline structure is insufficient or may not get best perfromance for more complex multiple functional unit pipelines with out-of-order issue and completion. For that, more complex control is necessary.

Example

1. **ADD R4, R2, R1**
2. **MUL R3, R4, R5**
3. **SUB R4, R6, R7**

Instruction 2 is dependent upon instruction 1 and cannot be issued until instruction 1 writes its result to R4. Instruction 3 has valid input operands and could be issued before instruction 2 (or even before instruction 1).

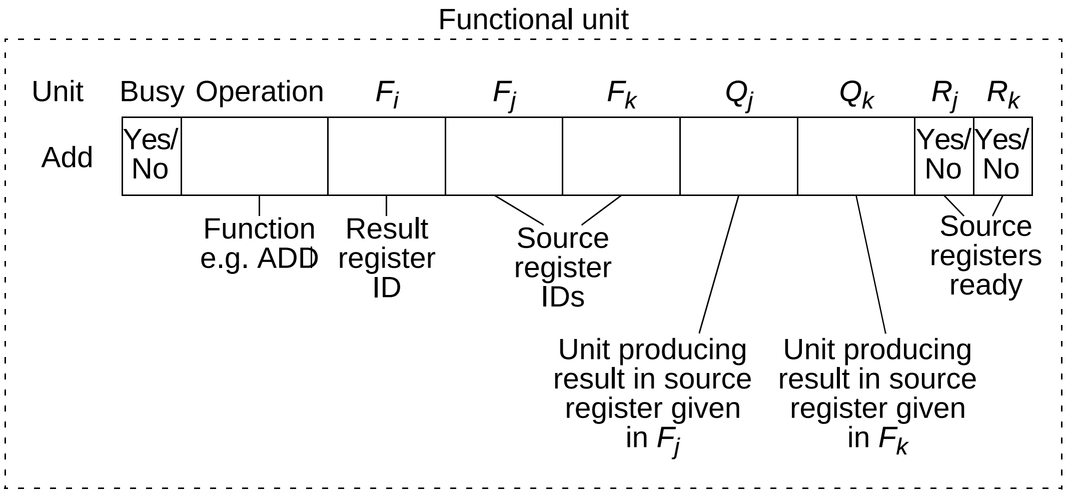
However, it must not write its result to R4 before instruction 1 writes its results to R4 and instruction 2 has read R4.

CDC 6600 Scoreboard (historical)

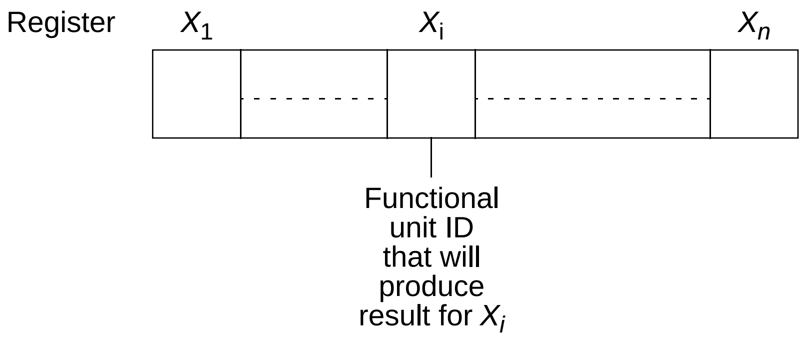
CDC 6600 computer system introduced the concept of a *scoreboard* which holds information to control when operands could be fetched and instruction execution can start and when results could be stored:

- If a structural hazard exists, for example a suitable functional unit is not available, the instruction is stalled.
- The instruction is also stalled if a write-after-write hazard exists, (a form of structural hazard - destination register being reused).
- Otherwise, the instruction is issued to a suitable function unit.

Operands are provided when available under the control of the scoreboard. Similarly, the scoreboard controls when results can be written (when write-after-read hazards are not present). Original CDC6600 scoreboard only of historical significance but devilishly difficult - so lets do it!.

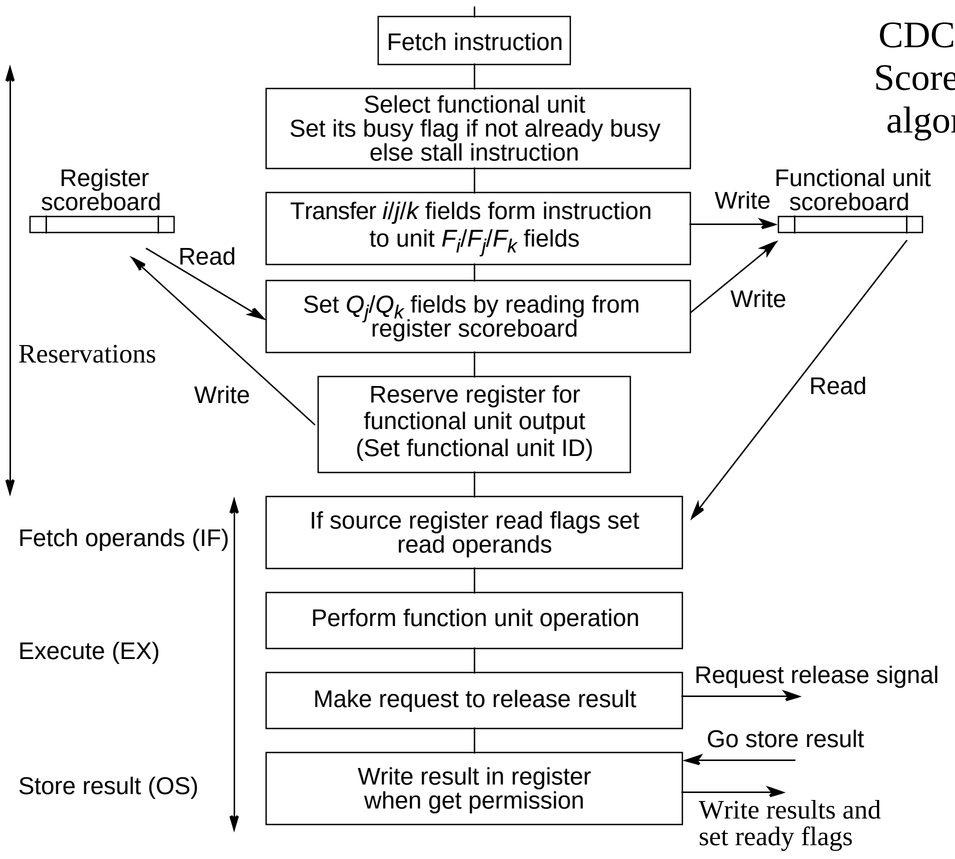


CDC 6600 Scoreboard information on functional unit



CDC 6600 Scoreboard information regarding source of register results

CDC 6600 Scoreboard algorithm


Fetch instruction
Select functional unit
Set its busy flag if not already busy
else stall instruction
Transfer i/j/k fields form instruction to unit $F_i/F_j/F_k$ fields
Set Q_j/Q_k fields by reading from register scoreboard
Reserve register for functional unit output (Set functional unit ID)
If source register read flags set read operands
Perform function unit operation
Make request to release result
Write result in register when get permission
Register scoreboard
Functional unit scoreboard
Read
Write
Write
Write
Read
Reservations
Fetch operands (IF)
Execute (EX)
Store result (OS)
Request release signal
Go store result
Write results and set ready flags

Interrupt handling

Term *interrupt* is the name given to a mechanism whereby the processor can stop executing its current program and respond to an event. This event could be within the processor or external to the processor. Requires that program being executed is stopped to execute interrupt service routine. After this routine executed, original program must be restarted.

When an interrupt occurs in a pipelined processor, instructions in the pipeline will be at various stages of completion.

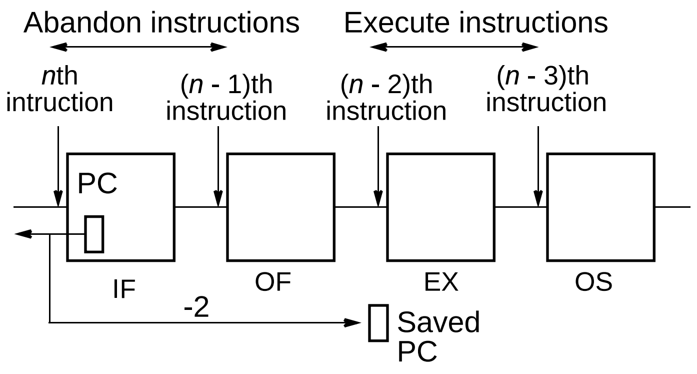
Interrupts can be handled as *precise interrupts* or *imprecise interrupts*.

Precise Interrupts

In *precise interrupts*, interrupt takes effect at an exact point within the pipeline. The following three conditions must be satisfied:

- All instructions issued prior to the instruction being interrupted are completely executed.
- All instructions issued after the instruction being interrupted are abandoned. The process state must not have been modified by these instructions.
- The interrupted instruction is either abandoned (without modify the process state) or allowed to be completely executed.

For a precise interrupt, sufficient information must to stored to enable the processor to restart at the exact point where it was interrupted.



Interrupt mechanism in pipeline

Imprecise interrupt

Precise interrupts can be very difficult and expensive to implement in a superscalar processor.

For an imprecise interrupt, not all information is stored to enable the processor to restart exactly where it was interrupted.

2000 Update

Processor design concepts of recent interest

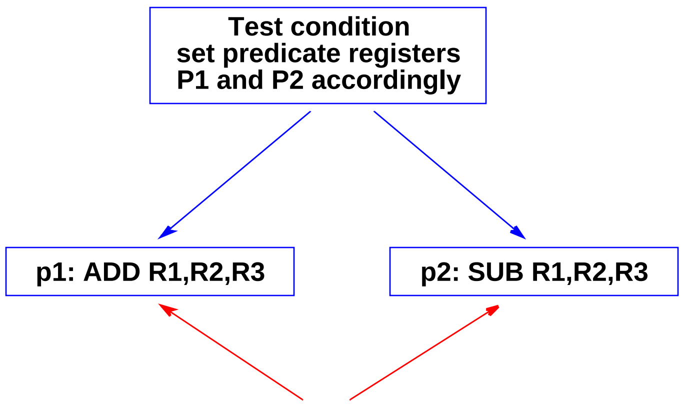
Branch Predication

Used in Intel IA-64/Merced processor (Intel’s new processor - not compatible with Pentium III). Proposed earlier by others (University of Illinois).

Used instead of traditional branch instructions (with branch prediction) to provide more opportunity for parallel execution. Replace branch instruction with an instruction which sets a “predicate” register to TRUE and another “predicate” register to FALSE. Then have “predicated” instructions - regular machine instructions but with add field which specifies which predicate register must be TRUE for the instruction to complete. Can start execution of the predicated instruction before that but it must not retire its results unless predicate is TRUE.

Example

```
if (condition) R1 = R2 + R3; else R1 = R2 - R3;
```



Start executing both instructions even before predicates set.

Only allow one to write to R1, the one in which the predicate is TRUE

Predicated code

```
CMPZ R1,R2,P1,P2   ;if R1>R1, set P1=TRUE, P2= FALSI
                   ;else set P1 = FALSE, P2 = TRUE
P1: ADD R1,R2,R3
P2: SUB,R1,R2,R3
```

P1/P2 are single bits which turn instruction on/off

Could have the predicate generators (CMPZ above) predicated itself.

(Actual Intel notation for predicate generator may be different.)

Advantages of predicated code

- Allows instructions to be executed simultaneously and “speculatively”
- Reduces branch misprediction penalties and hence can produced significantly faster code - Intel/HP quote 50% fewer branches and 37% faster code
- Most useful when branch prediction is hard to do accurately, e.g. in sorting, data compression, non-deterministic applications.

Instructions can be fetched/grouped together (see later)

Disadvantages

- Requires a completely new instruction set - cannot be fully grafted onto existing machines* - hence Intel’s completely new design.
- Speculatively executing instructions is wasteful of resources within the processor, if there is a high probability that the instructions will have to be abandoned

* Some existing processor do have “conditional move” instructions, but not full predication.

Speculative Load

Loading data from memory before it is needed to reduce the effects of memory latency. Done by moving the load instruction to earlier in the program than where it would normally be needed - “hoisted” to an earlier point.

Compilers can do this to some extent anyway.

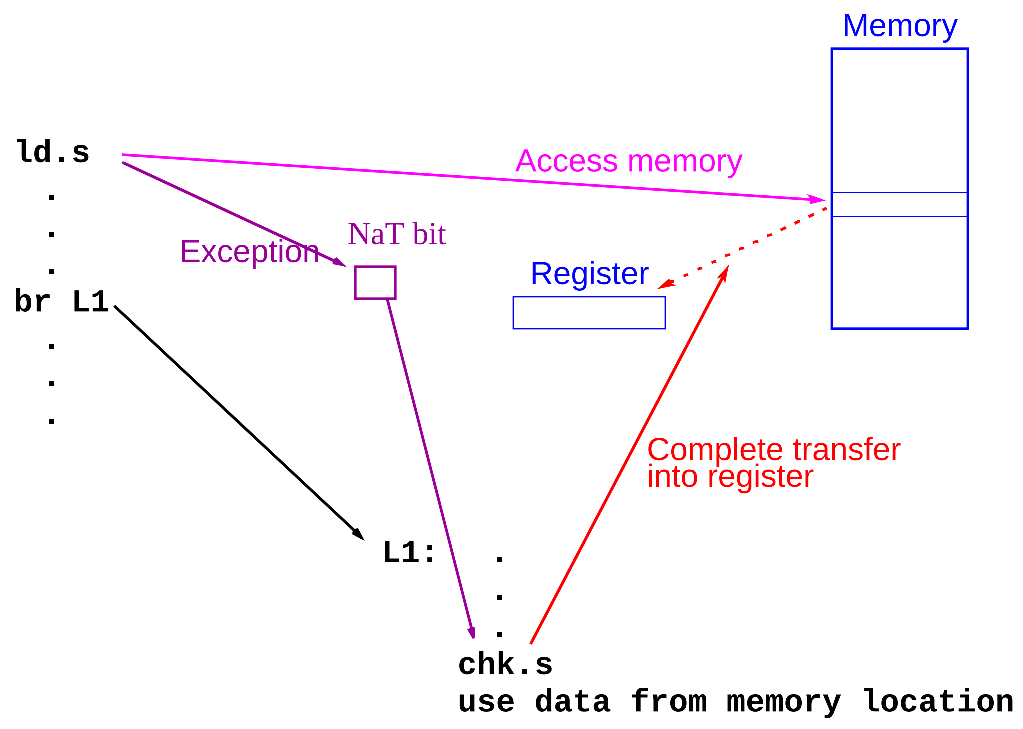
Problem occurs when the hoisting is across a branch instruction and a memory exception occurs, e.g. an invalid address, segmentation fault. This would generate an exception even if the load was not needed finally, i.e. the branch was down the path not needing the load.

IA-64 solution

Have two instructions:

- A speculative load instruction, **`ld.s`**, which performs the load operation without loading the destination register. If an exception occurs, a flag is set.
- A check instruction, **`chk.s`**, which checks whether an exception occurred. If it has, an exception handler is called, otherwise the destination is loaded.

The speculative load is placed as early as possible in the code. The check is placed where the result is needed. Check can be predicated.


Memory
ld.s
Access memory
NaT bit
Exception
Register
br L1
Complete transfer into register
L1:
chk.s
use data from memory location

Advantages of Speculative Load Instructions

- Hides the memory latency, a significant factor in obtaining improved performance. Intel quotes a 79% improvement when combined with predication (August et al, 1998)
- Particularly effective with many memory (cache) accesses such as in large databases, operating systems.
- Scheduling flexibility to obtain parallelism

Intel/Hewlett-Packard IA-64 Architecture

Based upon VLIW (very long instruction word) concept proposed in 1980's.

Independent instructions packaged into groups and sent toprocessor.

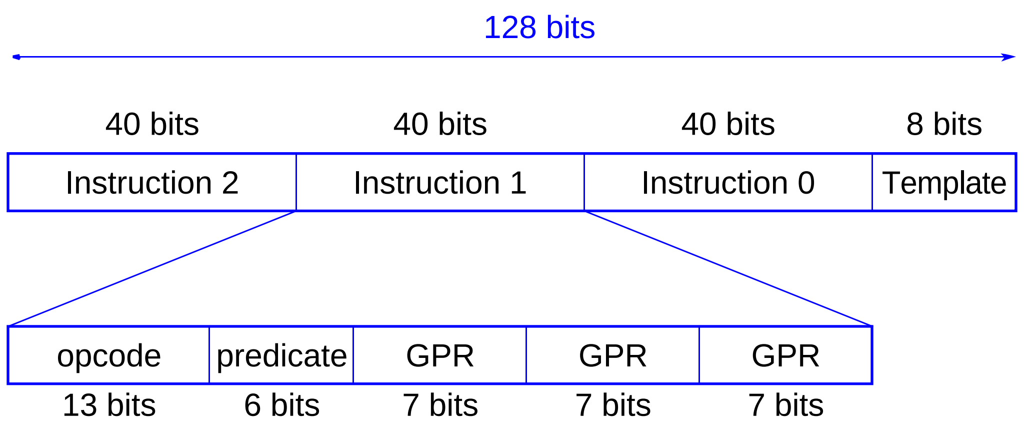
Processor executed group of instructions simultaneously (if sufficient internal resources available).

Instruction level parallelism where the compiler made the decision on which instructions were to be executed together.

Simple processor as it did not detect parallelism itself during execution.

Intel/HP call their version "EPIC - Explicit Parallel Instruction Computing."

IA-64 instruction format



GPR = specifies one of 128 general-purpose registers

Sources of further information

W.-M. Hwu, “Introduction to Predicated Execution,” *IEEE Computer*, January 1998, pp. 49-50.
M. S. Schlansker and R.R. Rau, “EPIC: Explicitly Parallel Instruction Computing,” *IEEE Computer,* February 2000, pp. 37-45.
C. Dulong, “The IA-64 Architecture at Work,” *IEEE Computer*, July 1998, pp. 24-32.
C. Zheng and C. Thompson, “PA-RISC to IA-64: Transparent Execution, No Recompilation,” *IEEE Computer*, March 2000, pp. 47-52. (see also other articles in this issue.)
“Inside Intel’s Mersed A Strategic Planning Discussion An Executive White Paper,” Aberdeen Group, Inc. Boston MA, July 1999. (See www.aberdeen.com)