# Chapter 8: Reduced Instruction Set Computer

## Reduced Instruction Set Computer (RISC)

In early 1980s a number of computer designers recommended that computers users use fewer instructions with a simple construct so that they can be executed much faster within the CPU without having to use memory as often. The concept of RISC architecture involves attempt to reduce the execution time by simplifying the instruction set of the computer. The major characteristics of RISC processor are:

1. Relatively few instructions.
2. Few and simple addressing modes.
3. Memory access limited to load and store instruction.
4. All operations done within the register of CPU.
5. Fixed length, easily decoded instruction format.
6. Single cycle instruction execution.
7. Hardwired rather than micro programmed control.

## Complex Instruction Set Computer (CISC)

The essential goal of a CISC architecture is to attempt to provide a single machine instruction for each statement that is written in high level language. A computer with large number of instruction is classified as Complex Instruction Set Computer (CISC). We have noted the trend to richer instruction sets which include a larger number of instructions and more complex instruction. Two principle reasons have motivated this trend, a desire to simplify compliers and desire to improve performance also the shift to HLL (High Level Language) so that now machines have to designed that provide better supports for HLLs. The translation from high level to machine level is done by compiler program. The task of a compiler is to generate a sequence of machine instruction for each high level language statement directly. The essential goal of a CISC architecture is to attempt to provide a single machine instruction for each statement is written in high level language.

The major characteristics of CICS architecture are:

1. Large number of instruction typically from hundreds to 250 instructions.
2. Some instruction that perform specialized tasks and are used infrequently.
3. A large variety of addressing modes typically from 5 to 20 different modes.
4. Variable length instruction formats.
5. Instruction that manipulate operands in memory.
6. Simple compilers will do the job.
7. Multiple cycle instruction (single HLL instruction is broken into smaller instruction).
8. Micro programmed implementation.
9. Simple control unit.
10. Indirect addressing required.

**Is CISC or RISC better?**

- Programs run faster on RISC processor than on CISC processor, however it may not be due to the RISC feature but because of technology, better compilers.
- Similar instruction set of RISC processor results in larger memory requirement compared to similar programs.
- Compilers for CISC processor, however cost of storage unit is being inexpensive.
- Most of the current processors are not typically RISC or CISC; they combine advantage of both approaches.

Comparison

| Architecture characteristics | CISC | RISC |
|---|---|---|
| - Instruction size | Varies | One size usually 32 bits |
| - Instruction format | Field placement varies | Regular, consistent placement of fields. |
| - Instruction semantics Semantics( gap between HLL to LLL) | Varies from simple to complex, possible many dependent operations per instruction | Almost always one simple operation. |
| - Registers | Few, sometimes special | Many general purpose |
| - Memory reference | Bundled with operations in many different types of instruction | Not bundled with operations i.e. load or store architecture. |
| - Hardware design | Exploit micro-coded implementations | Exploit implementation with one pipeline and no micro-code. |

**Instruction Pipelining/ Instruction level pipelining**

Pipelining is an implementation technique where multiple instructions are overlapped. Instruction is divided into no. of states so that at a clock cycle one stage of instruction is being processed, while another stage of another instruction will be processed in parallel, thus speeding the processing time. Pipelining does not decrease the time for individual instruction but at the same time it does a lot more stages/ jobs thus increasing the output.

$$speedup = \frac{cycle\ required\ during\ unpipeling\ execution}{cycle\ required\ during\ pipeling\ execution}$$

Suppose we have 5 instructions, if we process those 5 instructions sequentially say each instruction take 4 clock cycles then total clock cycles is 20. But if pipelining is done with 4 stages just 8 clock cycle will be sufficient. Speed up=20/8=2.5

| Clc cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| Instr. 1 | F | D | E | W | | | | |
| Instr. 2 | | F | D | E | W | | | |
| Instr. 3 | | | F | D | E | W | | |
| Instr. 4 | | | | F | D | E | W | |
| Instr. 5 | | | | | F | D | E | W |

Basic pipeline steps

1. Instruction fetch (IF): Instruction pointed to by pc is fetched from memory.
2. Instruction Decode (ID): Instruction decode.
3. Execution/ Effective address calculation (EX): the ALU operates on operand from ALU, input register & eventually puts the result in ALU register.
4. Memory access/Branch completion (MEM): only for load, store & branch instruction.
5. Write back (WB): the result of the instruction execution is stored into register file.

| IF | ID | EX | MEM | WB |
|---|---|---|---|---|

Types of pipelining

1. Simple pipeline: This is simple pipelining with single stage completed in single clock cycle.

| | | | | | |
|---|---|---|---|---|---|
| F | D | E | W | | |
| | F | D | E | W | |
| | | F | W | E | W |

2. Super pipeline: The next stage of instruction starts before the completion of first stage of instruction.

| | | | | | | |
|---|---|---|---|---|---|---|
| F | D | E | W | | | |
| | F | D | E | W | | |
| | | F | D | E | W | |

3. Super scalar pipeline: Number of same stages of different instruction is carried out at once.

| | | | | | |
|---|---|---|---|---|---|
| F | D | E | W | | |
| F | D | E | W | | |
| F | D | E | W | | |
| | F | D | E | W | |
| | F | D | E | W | |
| | F | D | E | W | |

## Arithmetic pipeline

Pipeline units are usually found in very high speed computers. They are used to implement for floating operations like multiplication. The floating point operations are easily decomposed into sub operations and then calculated. Example: a pipeline unit for floating addition and subtraction. Consider the two normalized floating point numbers

$X = 0.9504x10^3$ & Y$0.8200x10^2$

Now

Segment 1: Two exponents are subtracted 3-2=1. The larger exponent is chosen as the exponent of the result.

Segment 2: shift the mantissa of y to the right to obtain $X = 0.9504x10^3$ & Y$0.0820x10^3$. This aligns the mantissa under the same exponent.

Segment 3: the addition of two mantissa in segment 3 produces the sum Z$1.0324x10^3$.

Segment 4: the sum is adjusted by the normalizing the result so that it has a fraction with non zero first digit. Z$0.10324x10^4$.

The comparator, shifter, adder-subtractor, incrementer and decrementer in the floating point pipeline are implemented with the combination circuits, suppose the time delay of the four segments are t1=60ns, t2=70ns, t3=100ns & t4=80ns, and the interface register have a delay of tr=10ns,the clock cycle is chosen to be tp= t3+tr=110ns, and equivalent non pipeline floating adder-subtractor will have delay time of tn=t1+t2+t3+t4+t5+tr=320ns.

Speed up=320/110=2.9 over a non pipelined adder.

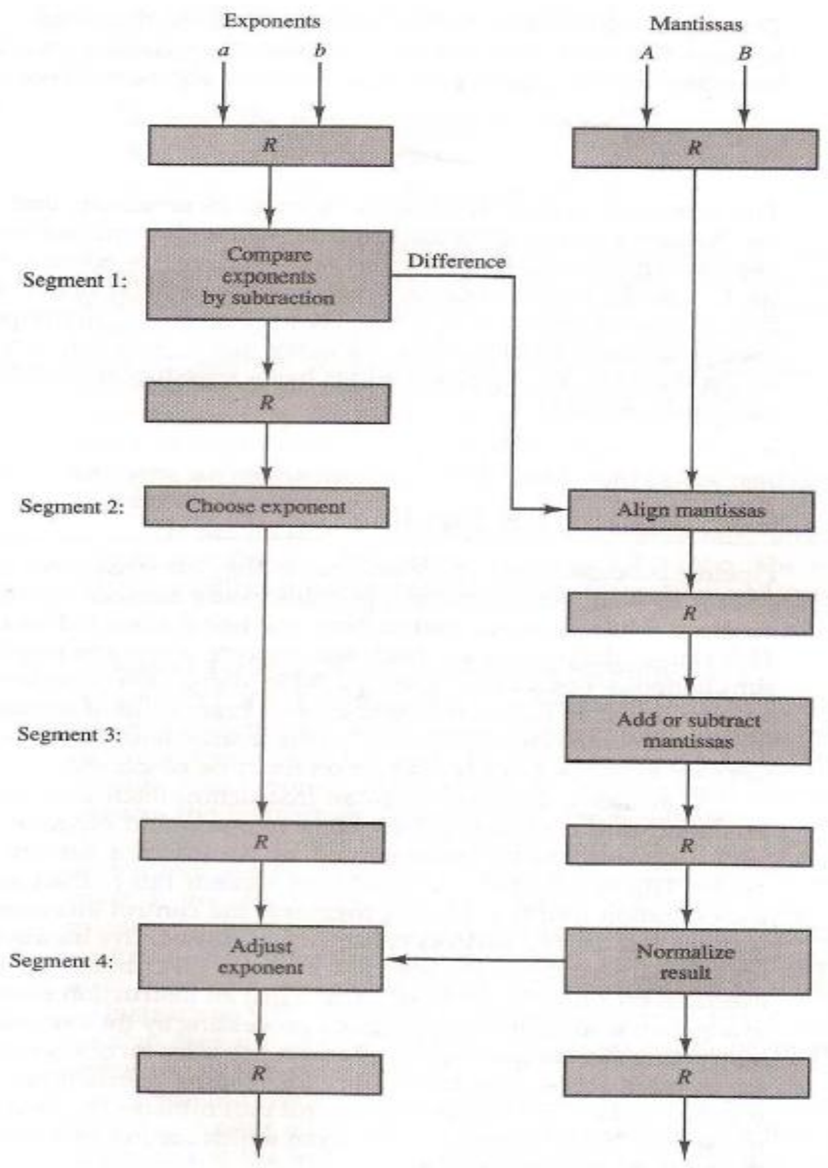

Figure: pipeline for floating point addition and subtraction

## Instruction pipeline

An instruction pipeline reads consecutive instruction from memory while previous instruction is being executed in other segments. These causes the instructions fetch and execute phases to overlap and perform simultaneous operations. Computers with complex instruction require other phases in addition to the fetch and execute to process an instruction completely. In the most general case the computers needs to process following sequence of steps.

1. Fetch the instruction into memory.
2. Decode the instruction.
3. Calculate the effective address.
4. Fetch the operand from memory.
5. Execute the instruction.
6. Store the results in proper place.

Certain difficulties that will prevent the instruction pipeline from operating at its maximum rate are different segments taking different time to operate on the incoming information. Two or more segments may require memory access at the same time causing one segment to wait until another is finished with memory.

Example

1. FI: is the segment that fetches an instruction.
2. DA: is the segment that decodes the instruction and calculates the effective address.
3. FO: is the segment that fetches the operand.
4. EX: is the segment that executes the instruction.

| Steps | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Instruction 1 | FI | DA | FO | EX | | | | | | | | | |
| 2 | | FI | DA | FO | EX | | | | | | | | |
| Branch 3 | | | FI | DA | FO | EX | | | | | | | |
| 4 | | | | FI | - | - | FI | DA | FO | EX | | | |
| 5 | | | | | - | - | - | FI | DA | FO | EX | | |
| 6 | | | | | | | | | FI | DA | FO | EX | |
| 7 | | | | | | | | | | FI | DA | FO | EX |

Figure: timing of instruction pipeline

**Pipeline hazards**

A pipeline hazards occurs when an instruction cannot complete a step in its execution, due to some events in previous clock cycle. When an instruction must be held for one or more clock pulse in-order to complete a step in its execution, this is called "pipeline stall" or "bubble".

**Type of hazards**

1. Structural hazards (resource conflicts ):

These hazards arises from resources conflicts when hardware cannot support all possible combinations of instruction in simultaneous overlapped execution e.g for unified cache memory

| Instruction | CC1 | CC2 | CC3 | CC4 | CC5 | CC6 | CC7 |
|---|---|---|---|---|---|---|---|
| Load | IF | ID | EX | MEM | WB | | |
| Instr 1 | | IF | ID | EX | MEM | WB | |
| Instr 2 | | | IF | ID | EX | MEM | WB |
| Instr 3 | | | | IF | ID | EX | MEM |
| Instr 4 | | | | | IF | ID | EX |

Both MEM and IF require to refer memory but we have unified cache where the instruction and data are stored in a single cache and in each cycle only one request can be processed.

Remedy

- Introduce stall.
- Replicating resources (increase resources).

2. Data hazards: These occur when one step in pipeline must wait the completion of a previous instruction, so there by creating conflict. A good compiler can reduce these hazards but not eliminate them completely.

Types

a. RAW (True dependency): A read after write hazard occurs when an instruction needs an operand whose value will be evaluated in the previous instruction so 2nd instruction must wait till the completion of 1st instruction.

   Add r1, r2, r3

   Mul r4, r1, r5

|  | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Add | IF | ID | EX | MEM | WB |  |
| Mul |  | IF | ID | EX | MEM | WB |

   Here the value of r1 is needed at cycle 4 but it is written at cycle 5

b. WAR (false / anti dependency): a write after read hazard is the reverse of RAW. Here first the write operation will occur before read, but there should be read operation first.

   Example:

   Mul r1, r2, r3

   Add r2, r4, r5

   If the add instruction is completed first then the mul instruction will get new value of r2, but it needs the old value of r2.

   Remedy: reordering

c. WAW: a write after hazard is a situation when j instruction tries to write an operand before it is written by i. the written up being performed in the wrong order , leaving the value written i, rather than value written by j in destination.

   Remedy: data forwarding

3. Control hazards

Most harmful hazard arises from the need to make decision based on the result on one instruction while others are executing. A typical e.g would be the execution of conditional branch instruction if the branch is taken the instruction currently in the pipeline might be invalid.

**Remedy**

1. Prefetch branch target:

- When the branch instruction is decoded, begin to fetch the branch target instruction and place in prefetch buffers.
- If branch is not taken, the sequential instruction is ready in the pipe.
- If branch is taken the next instruction has been prefetched and results in minimal branch delay.

2. Branch target buffer (BTB): The BTB is associative memory included in the fetch segment of pipeline. Each entry in BTB consists of the address previously executed branch instruction and target instruction for that branch. It also store next few instruction after branch target instruction.

   When pipeline decodes a branch instruction it searches it's associative memory in BTB for the address of instruction, if it is in BTB the instruction is available directly and prefetch continuous from the new path. If the instruction is not in BTB, pipeline shifts to new instruction stream and store the target instruction in BTB.

3. Loop buffer: a variation of BTb when a program loop is detected in the program it is stored in loop buffer in its entirely including all branches.
4. Branch prediction: A pipeline with branch prediction uses some additional logic to guess the outcome of a conditional branch instruction before it is executed. The pipeline then begins prefetching the instruction stream from the predicted path. A correct prediction eliminated the wasted time caused by branch penalties.
5. Delayed branch: A procedure employed in most RISC processor is delayed branch. In this procedure the compiler detects the branch instruction and re arranges the machine language code sequence by inserting useful instruction that keep the pipeline operating without interruptions. An example of delayed branch is insertion of no operation instruction after branch instruction. This causes the computer to fetch the target instruction during the execution of no operation instruction allowing a continuous flow of pipeline

## RISC pipeline

Instruction pipelining is often used to enhance performance. Let us reconsider this in the context of a RISC architecture. Most instructions are register to register, and an instruction cycle has the following two stages:

• I: Instruction fetch.

• E: Execute. Performs an ALU operation with register input and output.

For load and store operations, three stages are required:

• I: Instruction fetch.

• E: Execute. Calculates memory address

• D: Memory. Register-to-memory or memory-to-register operation.

Figure 13.6a depicts the timing of a sequence of instructions using no pipelining. Clearly, this is a wasteful process. Even very simple pipelining can substantially improve performance. Figure 13.6b shows a two-stage pipelining scheme, in which I and E stages of two different instructions are performed simultaneously. The two stages of the pipeline are an instruction fetch stage, and an execute/memory stage that executes the instruction, including register-to-memory and memory to- register operations. Thus we see that the instruction fetch stage of the second instruction can be performed in parallel with the first part of the execute/memory stage. However, the execute/memory stage of the second instruction must be delayed until the first instruction clears the second stage of the pipeline. This scheme can yield up to twice the execution rate of a serial scheme. Two problems prevent the maximum speed-up from being achieved. First, we assume that a single-port memory is used and that only one memory access is possible per stage. This requires the insertion of a wait state in some instructions. Second, a branch instruction interrupts the sequential flow of execution. To accommodate this with minimum circuitry, a NOOP instruction can be inserted into the instruction stream by the compiler or assembler. Pipelining can be improved further by permitting two memory accesses per stage. This yields the sequence shown in Figure 13.6c. Now, up to three instructions can be overlapped, and the improvement is as much as a factor of 3. Again, branch instructions cause the speedup to fall short of the maximum possible. Also, note that data dependencies have an effect. If an instruction needs an operand that is altered by the preceding instruction, a delay is required. Again, this can be accomplished by a NOOP. The pipelining discussed so far works best if the three stages are of approximately equal duration. Because the E stage usually involves an ALU operation, it may be longer. In this case, we can divide into two sub-stages:

E1: Register file read

E2: ALU operation and register write

Because of the simplicity and regularity of a RISC instruction set, the design of the phasing into three or four stages is easily accomplished. Figure 13.6d shows the result with a four-stage pipeline. Up to four instructions at a time can be under way, and the maximum potential speedup is a factor of 4. Note again the use of NOOPs to account for data and branch delays.

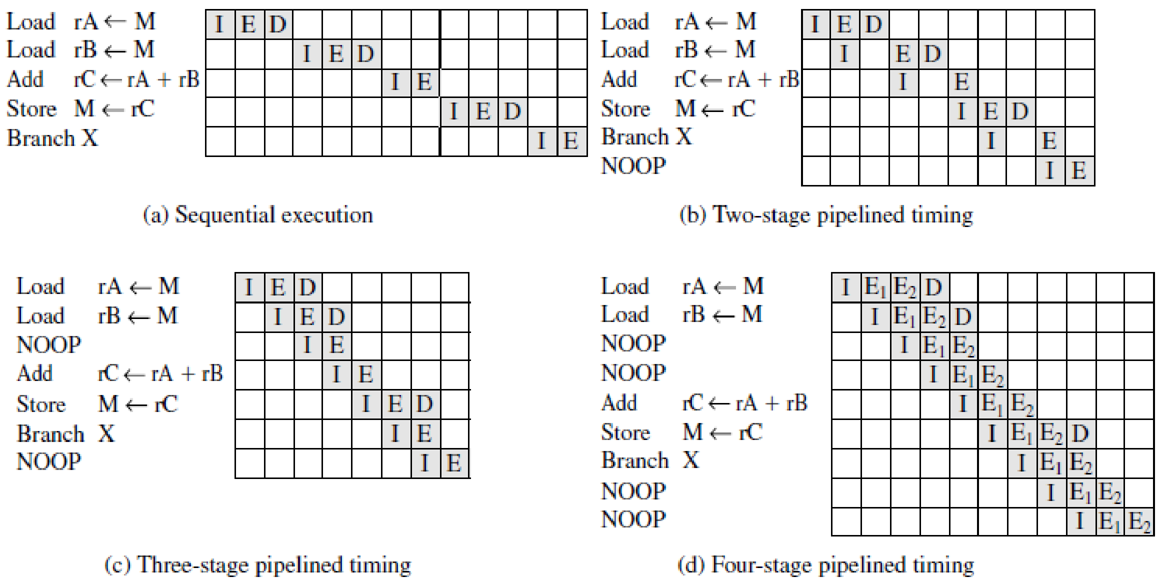

Figure 13.6 The Effects of Pipelining

## Optimization of Pipelining

Because of the simple and regular nature of RISC instructions, pipelining schemes can be efficiently employed. There are few variations in instruction execution duration, and the pipeline can be tailored to reflect this. However, we have seen that data and branch dependencies reduce the overall execution rate.

***DELAYED BRANCH*** To compensate for these dependencies, code reorganization techniques have been developed. First, let us consider branching instructions. *Delayed branch,* a way of increasing the efficiency of the pipeline, makes use of a branch that does not take effect until after execution of the following instruction (hence the term *delayed*). The instruction location immediately following the branch is referred to as the *delay slot*. This strange procedure is illustrated in Table 13.8. In the column labeled "normal branch," we see a normal symbolic instruction machine-language program. After 102 is executed, the next instruction to be executed is 105. To regularize the pipeline, a NOOP is inserted after this branch. However, increased performance is achieved if the instructions at 101 and 102 are interchanged. Figure 13.7 shows the result.

Table 13.8 Normal and Delayed Branch

| Address | Normal Branch | | Delayed Branch | | Optimized Delayed Branch | |
|---|---|---|---|---|---|---|
| 100 | LOAD | X, rA | LOAD | X, rA | LOAD | X, rA |
| 101 | ADD | 1, rA | ADD | 1, rA | JUMP | 105 |
| 102 | JUMP | 105 | JUMP | 106 | ADD | 1, rA |
| 103 | ADD | rA, rB | NOOP | | ADD | rA, rB |
| 104 | SUB | rC, rB | ADD | rA, rB | SUB | rC, rB |
| 105 | STORE | rA, Z | SUB | rC, rB | STORE | rA, Z |
| 106 | | | STORE | rA, Z | | |

2. ***DELAYED LOAD*** A similar sort of tactic, called the delayed load, can be used on LOAD instructions. On LOAD instructions, the register that is to be the target of the load is locked by the processor. The processor then continues execution of the instruction stream until it reaches an instruction requiring that register, at which point it idles until the load is complete. If the compiler can rearrange instructions so that useful work can be done while the load is in the pipeline, efficiency is increased.

3. **LOOP UNROLLING** Another compiler technique to improve instruction parallelism is loop unrolling. Unrolling replicates the body of a loop some number of times called the unrolling factor (u) and iterates by step u instead of step 1. Unrolling can improve the performance by

   • reducing loop overhead

   • increasing instruction parallelism by improving pipeline performance

   • improving register, data cache, or TLB locality

   Figure illustrates all three of these improvements in an example. Loop overhead is cut in half because two iterations are performed before the test and branch at the end of the loop. Instruction parallelism is increased because the second assignment can be performed while the results of the first are being stored and the loop variables are being updated. If array elements are assigned to registers, register locality will improve because a[i] and a [i+1] are used twice in the loop body, reducing the number of loads per iteration from three to two.

```
do i=2, n-1
       a[i] = a[i] + a[i-1] * a[i+1]
end do
```

**(a) Original loop**

```
do i=2, n-2, 2
       a[i] = a[i] + a[i-1] * a[i+1]
       a[i+1] = a[i+1] + a[i] * a[i+2]
end do

if (mod(n-2, 2) = i)  then
   a[n-1] = a[n-1] + a[n-2] * a[n]
end if
```

**(b) Loop unrolled twice**

Figure 13.8 Loop unrolling

**Register file**

We know that there are a large proportion of assignment statements in HLL programs, and many of these are of the simple form A←B, Also, there is a significant number of operand accesses per HLL statement. If we couple these results with the fact that most accesses are to local scalars, heavy reliance on register storage is suggested. The reason that register storage is indicated is that it is the fastest available storage device, faster than both main memory and cache. The complete set of register is known as ***register file*** and particular set of register is called ***window***. The register file is physically small, on the same chip as the ALU and control unit, and employs much shorter addresses than addresses for cache and memory. Thus, a strategy is needed that will allow the most frequently accessed operands to be kept in registers and to minimize register-memory operations.

Two basic approaches are possible, one based on ***software and the other on hardware***. The software approach is to rely on the compiler to maximize register usage. The compiler will attempt to allocate registers to those variables that will be used the most in a given time period. This approach requires the use of sophisticated program-analysis algorithms. The hardware approach is simply to use more registers so that more variables can be held in registers for longer periods of time.

## Register Windows

The use of a large set of registers should decrease the need to access memory. The design task is to organize the registers in such a fashion that this goal is realized. Because most operand references are to local scalars, the obvious approach is to store these in registers, with perhaps a few registers reserved for global variables. On every call, local variables must be saved from the registers into memory, so that the registers can be reused by the called program. Furthermore, parameters must be passed. On return, the variables of the parent program must be restored (loaded back into registers) and results must be passed back to the parent program. The solution is based on two forms. First, a typical procedure employs only a few passed parameters and local variables. Second, the depth of procedure activation fluctuates within a relatively narrow range. To exploit these properties, multiple small sets of registers are used, each assigned to a different procedure. A procedure call automatically switches the processor to use a different fixed-size window of registers, rather than saving registers in memory. Windows for adjacent procedures are overlapped to allow parameter passing. The concept is illustrated in Figure below. At any time, only one window of registers is visible and is addressable as if it were the only set of registers. The window is divided into three fixed-size areas. *Parameter registers* hold parameters passed down from the procedure that called the current procedure and hold results to be passed back up. *Local registers* are used for local variables, as assigned by the compiler. *Temporary registers* are used to exchange parameters and results with the next lower level (procedure called by current procedure).The temporary registers at one level are physically the same as the parameter registers at the next lower level. This overlap permits parameters to be passed without the actual movement of data. Keep in mind that, except for the overlap, the registers at two different levels are physically distinct. That is, the parameter and local registers at level J are disjoint from the local and temporary registers at level J 1. To handle any possible pattern of calls and returns, the number of register windows would have to be unbounded. Instead, the register windows can be used to hold the few most recent procedure activations. Older activations must be saved in memory and later restored when the nesting depth decreases. Thus, the actual organization of the register file is as a circular buffer of overlapping windows.

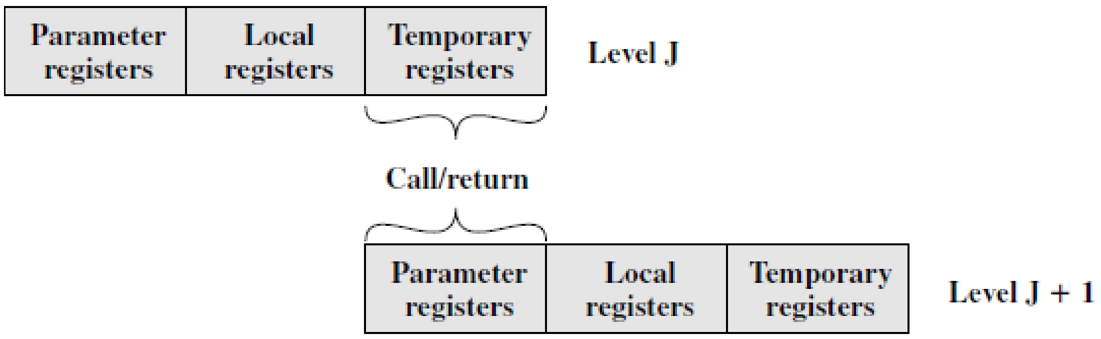

Figure: overlapping register window

## Register renaming

We have seen that during pipelining there is possibility of WAW dependencies and WAR dependencies. These dependencies differ from RAW data dependencies and resources conflicts, which reflect the flow of data through a program and the sequence of execution. WAW dependencies and WAR dependencies on the other hand arise because the value in register may no longer reflect the sequence of values dictated by the program flow.

When instruction is issued in sequence and complete in sequence it is possible to specify the contents of each register at each point in the execution. When out-of-order techniques are used the values in register cannot be fully known at each point in time just from consideration of the sequence of instruction dictated by the program. In effect values are in conflict for the use of register and the processor must restore those conflicts by occasionally stalling a pipeline stage.

One method for coping with these types of storage conflicts is based on a traditional resource conflict solution. i.e. duplication of resource. In this context the technique id referred to as ***register renaming.*** In essence register are allocated dynamically by the processor hardware and they are associated with the values needed by instruction at various points in time. When a new register value is created a new register is allocated for that value. Subsequent instruction that access that value as source operand in that register must go through a renaming process, the register reference in those instruction must be revised to refer the register containing the needed value. Thus the same original register reference in several different instructions may refer to different actual register, if different values are intended.

For example

| The instructions | can be written as |
|---|---|
| R7=R1*R2 | R7=R1*R2 |
| R1=R0-R2 | S1=R0-R2 |
| R3=R3*R1 | R3=R3*S1 |
| R1=R4+R4 | S2=R4+R4 |

HereS1, S2 are secreting register not visible to programmer. The register R1 is renamed to S1 in first part and to S2 in second part so that addition can be started before R1 is free.