CS311 Lecture: Pipelining, Superscalar, and VLIW Architectures

Last revised October 8, 2009

Objectives:

1. To introduce the basic concept of CPU speedup
2. To explain how data and branch hazards arise as a result of pipelining, and various means by which they can be resolved.
3. To introduce superpipelining, superscalar, and VLIW processors as means to get further speedup, including techniques for dealing with more complex hazard conditions that can arise.

Materials:

1. Pipelined MIPS Simulation; Interlocked MIPS Simulation
2. Handout showing stages
3. Sample programs: Add 1 to Memory 1000, Forwarding demo RType, Delayed Load Need Demo, Delayed Load Demo

I. Introduction

A. For any CPU, the total time for the execution of a given program is given by:

Time = cycle time * # of instructions * CP

$$= \frac{\#\text{ of instructions} * \text{CPI}}{\text{clock-rate}}$$

where: CPI (clocks per instruction) is the average number of clock cycles needed to execute an instruction

B. This equation suggests three basic strategies for running a given program in less time: (ASK CLASS TO MAKE SUGGESTIONS FOR EACH)

1. Reduce the cycle time (increase the clock rate)

   a) Can be achieved by use of improved hardware design/manufacturing techniques. In particular, reducing the FEATURE SIZE (chip area needed for one component), results in lower capacitance and nductance, and can therefore run the chip at a higher frequency.

   b) Do less computation on each cycle (which increases CPI, of course!)

2. Reduce the instruction count.

a) Better algorithms.

b) More powerful instruction sets - an impetus for the development of CISCs. (This, however, leads to increased CPI!)

3. Reduce CPI

a) Simplify the instruction set - an impetus for the original development of RISCs. (This, however, leads to increased program length!).

b) Do more work per clock. (This, however, requires a longer clock cycle and leads to a lower clock rate!).

4. In the case of clock rate and instruction count, there are speedup techniques that are clear "wins" - utilizing them does not adversely affect the other two components of the equation.

a) Improving the implementation so as to allow a faster clock.

b) Using more efficient algorithms

In both cases, however, it appears that we have reached a point where little further improvement is possible - CPU clock rates have maxed out in the 2-3 GHz range, and, in many cases, we have discovered virtually optimal algorithms.

5. In the case of CPI, it appears, that it is only possible to reduce CPI at the cost of more instructions or a slower clock. However, while there is no way to reduce the total number of clocks needed for an individual instruction without adversely impacting some other component of performance, it is possible to reduce the AVERAGE CPI by doing portions of two or more instructions in parallel. That is the topic we look at in the next few lectures.

C. There are several general ways to reduce average CPI

1. The first step in performing any instruction is fetching it from memory.

a) This step is always the same for any instruction (since we don't even know what it is until we've fetched it).

b) Moreover, some of the hardware used for fetching an instruction may be needed only at this point and then idle while the instruction is actually being executed.

c) Historically, one of the first speedup techniques to be used was pre-fetching of instructions.

The basic idea is this - while one instruction is executing, we fetch the next instruction from memory so that we can start decoding and executing it right away when the first instruction finishes. In effect, this reduces the average CPI by the number of clocks needed for instruction fetch.

Example: suppose the average CPI for some system were 5, of which one was used for instruction-fetch. Prefetching would reduce the average CPI to 4 - a 20% improvement.

2. We can go further along these line. If we look at the execution of a single instruction, we typically find that it uses other parts of the system at different times as well.

a) Example: It turns out that any instruction on mips only uses the ALU proper on its second cycle.

b) We could potentially reduce average CPI by having different instructions using different parts of the system at the same time - e.g. on mips one instruction might use the ALU at the same time other instructions are doing something else.

c) This yields a strategy known as PIPELINING.

3. We might achieve even greater parallelism by replicating portions of the system - e.g. if it had two ALU's it could do cycle 2 of two different instructions at the same time. This leads to various strategies including VLIW and Superscalar Architectures.

4. Pursuing this even further, we can replicate complete CPU's - or at least CPU cores- a strategy that is being pursued today in terms of dual or quad core CPUs

D. We will look at first three of these in turn. We will not consider now (though we will consider later) how overall system speed can be improved by using multiple CPU's.

II. Prefetching of Instructions

A. Suppose we have a CPU that executes an instruction in 4 distinct steps, each requiring one clock: fetch the instruction, and then (up to) 4 steps to execute it. (CPI = 4) Of course, on CISCS the number of steps per instruction varies widely. Even on a RISC, not all instructions need all the execution steps.

The timing of the CPU operations on such a machine can be pictured as follows, using a Gantt chart (where S1, S2, S3 are a series of instructions being executed one after the other):

```
Step 4              -S1-            -S2-            -S3-

Step 3         -S1-            -S2-            -S3-

Step 2     -S1-            -S2-            -S3-

Fetch  -S1-            -S2-            -S3-

Time ---------->
```

B. Prefetching would change the picture to this;

```
Step 4              -S1-        -S2-        -S3-

Step 3         -S1-        -S2-        -S3-

Step 2     -S1-        -S2-        -S3-

Fetch  -S1-        -S2-        -S3-

Time ---------->
```

1. Though each instruction still takes 4 cycles from start to finish, the average number of clocks per instruction - in the steady state - is three - i.e. one instruction completes every third cycle. This represents a 25 speedup.

2. Prefetching may not seem possible on machines that have variable length instructions, since we don't know how long an instruction is until we have decoded it (and perhaps partially executed it.). However,

many variable length instruction machines still do a form of prefetching by using an instruction queue that simply holds as yet uninterpreted bytes from the instruction stream.

a) Whenever execution calls for another byte from the instruction stream, one is taken from the queue if possible.

b) Whenever the memory is idle, if there is room in the queue then one or more bytes are fetched ahead.

c) Of coures, if the outcome of a conditional branch sends the program down a different path from the one the CPU has been prefetching on, then he queue is flushed.

III. Pipelining

A. The idea of prefetching can be extended to overlap other parts of instruction execution as well.The ultimate degree of parallelism between instructions would be total overlap between all phases of different instructions - e.g.

```
Step 4              -S1--S2--S3-

Step 3         -S1--S2--S3-

Step 2     -S1--S2--S3-

Fetch  -S1--S2--S3-

Time ---------->
```

[We have now reduced average per instruction time by 75% compared to the original scheme]

1. We call this a FULLY PIPELINED CPU. In the steady state, it completes one instruction on every cycle, so its average CPI is 1. This is, in fact, what RISC implementations do - and RISC ISAs are structured to make such an implementation straightforward.

2. Of course, an average CPI of 1 is attainable only when the pipeline is full of valid instructions. If the pipeline must be flushed (as would be the case whenever a branch is taken in program execution), it will take several cycles for the pipeline to fill up again.

B. Rather than discussing an actual MIPS pipeline, which uses both the rising and falling edge of the clock so that some stages require only half a clock period, we will discuss a simplified pipeline that is a bit easier to understand. It has four stages.

SHOW Pipelined MIPS simulation. Distribute handout of stage structure

Note that, though some hardware units appear in more than one stage, each data path exists in exactly one stage.

1. Stage 0 is instruction fetch. This step is, of course, the same for all instructins since we don't have the instruction yet.

   Each of the remaining stages has an Instruction Register. Stage 0 fetches an instruction into the Stage 1 IR, where execution of it begins. On each clock, the Stage 1 Instruction is moved to Stage 2 and the Stage 2 instruction is moved to Stage 3.

   DEMO: Load "Add 1 to Memory 1000" and clock a few times, showing how first iinstruction progresses through the stages.

2. Stage 1 does two things

   SHOW Tab

   a) It updates the PC. Depending on the Instruction in the IR, this may come directly from the instruction (J-Format) or by adding the Immediate field of the instruction to the PC (beq, bne taken) or by adding 4 to the current PC (most instructions including beq, bne not taken)

   b) It loads values into the ALU holding registers. One value always comes from the register selected by the rs field of the Stage 1 instruction. The other either comes from the register selected by rt or the Immediate field of the Stage 1 instruction. (A value is always loaded into both registers, even though one or both may turn out to be unused later)

3. Stage 2 performs an ALU operation on the values in the holding registers (which were put there by Stage 1 of the instruction) and stores a result into the ALU output register. What operation is performed is determined by the instruction in the Stage 2 IR. (Add for load/store; instructions like j, beq, bne don't actually need an ALU operation but one is done anyway, though the result is not used.)

4. Stage 3 stores a result, if appropriate, based on the instruction in the Stage 3 IR.

   a) For R-Type instructions, the ALU output is stored in a register.

   b) For load instructions, a memory location is stored in a register.

   c) For store instructions, a register is stored in memory

   d) For other instructions, nothing happens in this Stage.

5. Thus, four instructions are in the pipeline at any time - one at each stage. Although it takes up to 4 clock cycles to execute any one instruction, one instruction is completed on each clock and so the effective time for an instruction is only one cycle - a fourfold speedup.

C. However, overlapping instructions can lead to various sorts of hazards due to inter-instruction dependencies (hazards). Some of these arise, in fact, even with simple prefetching; but full pipelining makes them more of an issue.

1. One sort of hazard is called a BRANCH HAZARD or CONTROL HAZARD. It can arise if the current instruction is a branch (or jump).

   a) In this case, how can we know while the branch instruction is being executed ins Stage 1 whether to fetch the next sequential instruction or the one at the branch address in Stage 0? (Indeed, even if the branch is unconditional, how can one know what the branch address is if it involves computation as with relative mode addressing?)

   b) With simple prefetching, this problem could be avoided by ensuring that a conditional branch instruction does not require the final step (so the decision and target address are known before the next instruction needs to be fetched). However, with full pipelining this will not solve the problem.

   c) One possibility is to suspend (pre)fetching of instructions during execution of a branch instruction until the outcome is known and/or the target address is calculated. In the case of full pipelining, this is sometimes called a PIPELINE BUBBLE or STALL, and is implemented by changing the instruction that was fetched to something like a nop which effectively nullifies it. Of course, this has a negative impact on speed, sometimes called a “branch penalty”.

d) Some CPU's use some approach for "guessing" which way the branch will turn out. This is called BRANCH PREDICTION. How can such a prediction be done?

(1) One way to do the prediction is to use the following rule of thumb: assume that forward conditional branches will not be taken, and backward conditional branches will be taken.

Why? ASK

- Forward branches typically arise from a construct like

```
if something ...
        common case
else
        less common case
```

- Backward branches typically result from loops - and only the last time the branch is encountered will it not be taken.

(2) Some machines incorporate bits into the format for branch instructions whereby the compiler can furnish a hint as to whether the branch will be taken.

(3) Some machines maintain a branch history table which stores the address from which a given branch instruction was fetched and an indication as to whether it was taken the last time it was executed

(4) Of course, if a branch is predicted to be taken, the problem remains of knowing the target address if it is computed during execution of the instruction. On a pipelined CPU that uses a branch history table, this can be addressed by storing the actual target address as well - if relative mode addressing is used, the target for a given branch instruction will always be the same.

e) Some machines (such as MIPS) _always_ execute the instruction after the branch instruction, even if the branch is taken. This is called DELAYED BRANCH.

(This is the reason for the assembler inserting a nop after a branch instruction. In many cases - as we shall see later - it turns out to be possible to rearrange instructions in such a way as to place a useful instruction in this slot)

f) Of course, other control transfer instructions - e.g. j, jal face a similar problem, even though the branch is always taken, since the target address must still be computed. Machines (like MIPS) that use delayed branch typically use this approach for these instructions as well.

2. With full pipelining, an additional complication arises.

a) Suppose we have the sequence of instructions S1, S2 ...; instruction S2 uses some register as one of its its operands, and suppose that the result of S1 is stored in this same register - e.g.

```
S1:     addi $1, $0, 0x2a
S2:     add  $2, $1, $0
```

(Clearly, the intention is for S2 to use the value stored by S1)

If S2 loads this register into an ALU input register in its Stage 1, and S1 doesn't store its result into this register until its Stage 3 (which coincides with S1's Stage 2) , then the value that S2 uses will be the PREVIOUS VALUE in the register - not the one stored by S1, as intended by the programmer.

This can be seen more clearly in the following

	time 0	time 1	time 2	time 3	time 4
S1: addi $1, $0, 0x2a	Stage 0: fetched	Stage 1: put 0 and 42 into ALU input registers	Stage 2: Compute ALU Output = 0 + 42	Stage 3: Store 42 in $1	
S2: addi $2, $1, $0		Stage 0: fetched	Stage 1: put $1 and $0 into ALU input registers	Stage 2: Compute ALU Output = sum of inputs	Stage 3: Store result in $2

Note how, at time 2, S2 uses the value in $1 before S1 stores its computed value at time 3!

b) This sort of situation is called a DATA HAZARD or DATA DEPENDENCY.

c) In this particular case, it can be resolved by a strategy called DATA FORWARDING. Observe that when an RType instruction is immediately followed by some other instruction that uses its result, the source value needed by the second instruction is being computed in the ALU - it just hasn't yet been placed into the ALU holding register or ultimately the correct register in the register file. (In the example, at time 2).

In such a case, the hardware can detect this situation and forward the value directly from the ALU output to the ALU input register (while also storing it into the output holding register on the next clock so it is available to future instructions. This produces a result like the following:

	time 0	time 1	time 2	time 3	time 4
S1: addi $1, $0, 0x2a	Stage 0: fetched	Stage 1: put 0 and 42 into ALU input registers	Stage 2: Compute ALU Output = 0 + 42	Stage 3: Store 42 in $1	
S2: addi $2, $1, $0		Stage 0: fetched	Stage 1: put forwarded 42 and 0 into ALU input registers	Stage 2: Compute ALU Output = 42 + 0	Stage 3: Store 42 in $2

Demo: (Using pipelined implementation)

load Forwarding Demo RType.
Be sure r1 contains zero.
clock twice
Note that Stage 1 IR contains rs = $1, and $1 is still zero
Note use of forwarding for rs.
clock again - note correct value loaded into holding register

d) However, this doesn't work for memory load instructions. A read from memory is not done until Stage 3 of the pipeline, which coincides with Stage 1 of the instruction two behind it. (Stage 2 is used to compute the address of the value to load)

Consider the following program. (Assume memory address 1000 contains 42)

```
S1:     lw   $1, 0x1000($0)
S2:     add  $2, $1, $0
```

This program executes as follows:

	time 0	time 1	time 2	time 3	time 4
S1: lw $1, 1000($0)	Stage 0: fetched	Stage 1: put 0 and 1000 into ALU input registers	Stage 2: Compute address = 0 + 1000	Stage 3: Load $1 with value at address 1000	
S2: addi $2, $1, $0		Stage 0: fetched	Stage 1: put $1 and $0 into ALU input registers	Stage 2: Compute ALU Output = sum of inputs	Stage 3: Store result in $2

Demo: (Using pipelined implementation)

load Delayed Load Need Demo
Be sure r1 contains zero.
clock twice
Note that Stage 1 IR contains rs = $1, and $1 is still zero
Note value has not yet been read from memory.
clock again - note incorrect value loaded into holding register

(1) .In such cases, one possible solution is to use an approach known as DELAYED LOAD: code may not use a register in the instruction immediately after one that loads it (If an instruction just after a load does try to use the same register as a source, something bad happens like it gets the OLD value.) Our program would need to incorporate a nop as was done for delayed branch:

```
S1:     lw      $1, 0x1000($0)
S2:     nop
S3:     add     $2, $1, $0
```

(a) This was the approach used by early MIPS implementations - the instruction immediately after a load may not use the register that was just loaded.

(b) It appears, from the above, that a delay of 2 cycles would actually be needed, but this was reduced to 1 by forwarding the value fetched from memory at time 3 into an ALU input register at time 1 of the instruction two behind. This produces the following result

	time 0	time 1	time 2	time 3	time 4	time 5
S1: lw $1, 1000($0)	Stage 0: fetched	Stage 1: put 0 and 1000 into ALU input registers	Stage 2: Compute address = 0 + 1000	Stage 3: Load $1 with value at address 1000 (42)		
S2: nop		Stage 0: fetched	[Do nothing]	[Do nothing]	[Do nothing]	
S3: addi $2, $1, $0			Stage 0: fetched	Stage 1: put forwarded value read (42) and 0 into ALU input registers	Stage 2: Compute ALU Output = 42 + 0	Stage 3: Store 42 in $2

Demo: (Using pipelined implementation)

load Delayed Load Demo
Be sure r1 contains zero.
clock three times
Note that Stage 1 IR contains rs = $1, and $1 is still zero
Note use of forwarding for rs.
clock again - note correct value loaded into holding register

(2) Another approach to handle such a situation is variously known as interlocking, a pipeline stall, or a "bubble".

(a) The hardware can detect the situation where the Stage 2 IR contains an instruction which will load a value into the same register as one of the source operands of the instruction in the Stage 1 IR. (This is a simple comparison between IR field contents that is easily implemented with just a few gates.)

(b) In such cases, the hardware can replace the instruction in the Stage 1 IR with a NOP and force Stage 0 to refetch the same instruction instead of going on to the next, and use forwarding for the value read from memory. This produces a result like the following

	time 0	time 1	time 2	time 3	time 4	time 5
S1: lw $1, 1000($0)	Stage 0: fetched	Stage 1: put 0 and 1000 into ALU input registers	Stage 2: Compute address = 0 + 1000	Stage 3: Load $1 with value at address 1000		
S2: addi $2, $1, $0		Stage 0: fetched	[Changed to NOP]	[Do nothing]	[Do nothing]	
				Stage 1: put forwarded value read and 0 into ALU input	Stage 2: Compute ALU Output = 42 + 0	Stage 3: Store 42 in $2

Demo: (Using interlocked implementation)

load Delayed Load Need Demo (that didn't work before).
Be sure r1 contains zero.
clock twice
Note that Stage 1 IR contains rs = $1, and Stage 2 IR rt = $1
Note that PC and Stage 1 IR are locked - i.e. won't change on next clock [note values]
Note that Stage 2 IR is marked to become a bubble [note value]
Clock again
Note that PC and IR 1 values are unchanged; IR 2 has become a nop; but value in IR2 has moved on to IR3
Note use of forwarding for rs.
clock again - note correct value loaded into holding register
Note how value being read from memory is forwarded to Stage 1 rs
Clock again
Note how correct value (42) has been loaded into first ALU input register

(c) Of course, interlocking means wasting a clock cycle when necessary, since the NOP does no useful work - but this is no more wasteful in terms of execution time than using delayed load, and the overall program is shorter.

(d) Later MIPS implementations (starting with MIPS ISA III) avoided the need for delayed load by using interlocking.

D. Because a RISC pipeline is so regular in its operation, the compiler may be expected to use knowledge about the pipeline ensure correct code or even to optimize the code it generates.

1. We've just noted that one problem faced by pipelined CPU's is data dependencies that cannot be resolved in the hardware by forwarding.

a) If the CPU uses delayed load, we can require the compiler to avoid emitting an instruction that uses the result of a load instruction immediately after that instruction. (The compiler can either put some other, unrelated instruction in between, or it can emit a NOP if all else fails.)

Example: suppose a programmer writes:

d = a + b + c + 1

This could be translated to the following:

```
lw     $10, a          # r10 <- a
lw     $11, b          # r11 <- b
nop    # inserted to deal with data hazard
add    $11, $10, $11   # r11 <- a + b
lw     $12, c          # r12 <- c
nop    # inserted to deal with data hazard
add    $12, $11, $12   # r12 <- a + b + c
add    $12, $12, 1     # r12 <- a + b + c + 1
sw     d, $12          # d <- a + b + c + 1
```

b) A "smart" compiler can sometimes eliminate the NOPS by rearranging the code. For example, the computation of d = a + b + c + 1 might be done by

```
lw    $10, a          # r10 <- a
lw    $11, b          # r11 <- b
lw    $12, c          # r12 <- c
add   $11, $10, $11   # r11 <- a + b
add   $12, $11, $12   # r12 <- a + b + c
add   $12, $12, 1     # r12 <- a + b + c + 1
sw    $12, d          # d <- a + b + c + 1
```

c) What if the hardware incorporates interlocks to stall the pipeline if an attempt is made to use a register that is currently being loaded, instead of using a delayed load?

(1) Interlocking becomes necessary in cases where the amount of parallelism in the system makes it unreasonable to require that the compiler anticipate all eventualities. (For example, this is the reason why MIPS implementations since MIPS ISA III have used interlocking, though most earlier MIPS implementation required the compiler to prevent the problem.)

(2) Rearranging code as above will still result in faster execution, since stalls will not be needed, so a "smart" compiler may attempt to do so to the extent possible.

2. Another source of potential problems - which we have already noted - is branch dependencies.

a) The delayed branch approach used by RISCs like mips must be allowed for by the compiler.

(1) All control transfer instructions (subroutine calls and returns as well as jumps) take effect AFTER the next instruction in sequence is executed, so such instructions are often followed by a nop, as we have seen in lab.

(2) However, a good compiler can often work with this feature of the hardware by inserting a branch instruction ahead of the last instruction to be done in the current block of code.

Consider the following example: Suppose we were compiling

```
if (x == 0)
        a = a + 1;
else
        a = a - 1;
```

(With x and a local variables allocated to reside in $15 and $16, respectively.) This could be translated as

```
        bne  $15, $0, else_part   # branch if x != 0
        nop
        addi $16, $16, 1          # a = a + 1
        b    end_if
        nop
else_part:
        addi    $16, $16, -1      # a = a - 1
end_if:
```

We could eliminate one nop as follows:

```
        bne  $15, $0, else_part   # branch if x != 0
        nop
        b    end_if
        addi $16, $16, 1          # a = a + 1
else_part:
        addi    $16, $16, -1      # a = a - 1
end_if:
```

(3) Unfortunately, in this case the first nop cannot be eliminated, But suppose the original code were

```
if (x == 0)
        a = a + 1;
else
        a = a - 1;
b = b + 1;
```

(With b a local variable in $17)

How would it be possible to translate this code without using any nops?

ASK

```
      bne  $15, $0, else_part   # branch if x != 0
      addi $17, $17, 1          # b = b + 1
      b    end_if
      addi $16, $16, 1          # a = a + 1
else_part:
      addi    $16, $16, -1      # a = a - 1
end_if:
```

E. The potential speedup from pipelining is a function of the number of stages in the pipeline.

1. For example, suppose that an instruction that would take 4 ns to execute is implemented using a 4 stage pipeline with each stage taking 1 ns. Then the speedup gained by pipelining is

   w/o pipeline - 1 instruction / 4 ns
   with pipeline - 1 instruction / 1 ns 4ns/1ns = 4:1

   Now if, instead, we could implement the same instruction using a 5 stage pipeline with each stage taking 0.8 ns, we could get a 5:1 speedup instead.

2. This leads to a desire to make the pipeline consist of as many stages as possible, each as short as possible. This strategy is known as SUPERPIPLINING.

   a) The basic idea is to break the execution of an instruction into smaller pieces and use a faster clock, perhaps performing operations on both the falling and the rising edge of the clock (i.e. having two pipeline stages per clock.)

b) Of course, the longer the pipeline, the greater the potential waste of time due to data and branch hazards.

(1) Branch hazards can be reduced by doing the relevant computations in the earliest possible pipeline stage, or by using a branch history table (with saved target addresses), or by reducing the need for branches through a technique known as predication - to be discussed below.

(2) Data hazards, again, may lead to a need to use interlocking to ensure correct results; with the order of operations in code driven by minimizing the need for this.

c) Note that superpipelining attempts to maintain CPI at 1 (or as close as possible) while using a longer pipeline to allow the use of a faster clock.

F. So far in our discussion, we have assumed that the time for the actual computation portion of an instruction is a single cycle (the rest of the steps being used to handle getting the operands and storing the result) This is realistic for simple operations like AND, fixed point ADD etc. However, for some instructions multiple cycles are needed for the actual computation.

1. These include fixed-point multiply and divide and all floating point operations.

2. To deal with this issue, some pipelined CPU's simply exclude such instructions from their instruction set - relegating them to co-processors or special hardware (e.g. MIPS approach).

3. If such long operations are common (as would be true in a machine dedicated to scientific computations), further parallelism might be considered in which the computation phases of two or more instructions overlap. We will not discuss this now, but will come back to it under vector processors later in the course.

IV. Moving Beyond Basic Pipelining By Replicating Functional Units

A. It would appear, at first, that a CPI of 1 is as good as we can get - so there is nothing further that can be done beyond full pipelining to reduce CPI. Actually, though, we can get CPI less than one if we execute two or more instructions fully in parallel (i.e. fetch them at the same time, do each of their steps at the same time, etc) by duplicating major portions of the instruction execution hardware.

1. If we can start 2 instructions at the same time and finish them at the same time, we complete 2 instructions per clock, so average CPI drops to 0.5. If we can do 4 at a time, average CPI drops to 0.25.

2. In describing architectures like this, the term ISSUE is used for starting an instruction and RETIRE is used for completing an instruction.

   a) Because not all instructions require the same number of clock cycles, a system may actually retire a greater or lesser number of instructions on any clock than it issues on that clock, but the average number of issues/retires per clock will be the same.

   b) Various hazards make it impossible to always achieve the maximum degree of parallelism. Thus, in some cases the machine will issue fewer instructions on a clock than it potentially could (perhaps even zero). When an instruction is not issued on some clock due to a hazard, it is held until the next clock, when an attempt is made again to issue it.

3. Multiple issue is facilitated by taking advantage of the fact that many CPU's have separate execution units for executing different types of instructions - e.g. there may be:

   a) An integer execution unit used for executing integer instructions like add, bitwise or, shift etc.

   b) A floating point execution unit for executing floating point arithmetic instructions. (Note that many architectures use separate integer and floating point register sets).

   c) A branch execution unit used for executing branch instructions.

   (etc)

d) If two instructions need two different execution units (e.g. if one is an integer instruction and one is floating point) then they can be issued simuiltaneously and execute totally in parallel with each other, without needing to replicate execution hardware (though decode and issue hardware does need to be replicated.)

Note that, for example, many scientific programs contain a mixture of floating point operations (that do the bulk of the actual computation), integer operations (used for subscripting arrays of floating point values and for loop control), and branch instructions (for loops). For such programs, issuing multiple instructions at the same time becomes very feasible.

4. The earliest scheme used for doing this was the VERY LONG INSTRUCTION WORD architecture. In this architecture, a single instruction could specify more than one operation to be performed - in fact, it could specify one operation for each execution unit on the machine.

a) The instruction contains one group of fields for each type of instruction - e.g. one to specify an integer operation, one to specify a floating point operation, etc.

b) If it is not possible to find operations that can be done at the same time for all functional units, then the instruction may contain a NOP in the group of fields for unneeded units.

c) The VLIW architecture required the compiler to be very knowledgeable of implementation details of the target computer, and may require a program to be recompiled if moved to a different implementation of he same architecture.

d) Because most instruction words contain some NOP's, VLIW programs tend to be very long.

5. Current practice - is to use SUPERSCALAR architecture.

a) A superscalar CPU fetches groups of instructions at a time - typically two (64 bits) or four (128 bits) and decodes them in parallel.

b) That is, a superscalar CPU has just one instruction fetch unit, but it fetches a whole group of instructions, but it has 2 or 4 decode units and a number of different execution units.

c) If the instructions fetched together need different execution units, then they are issued at the same time. If two instructions need the same execution unit, then only the first is issued; the second is issued on the next clock. (This is called a STRUCTURAL HAZARD).

d) To reduce the number of structural hazards that occur, some superscalar CPU's have two or more integer execution units, along with a branch unit and a floating point unit, since integer operations are more frequent. Or, they might have a unit that handles integer multiply and divide and one that does add and subtract.

6. Once again, the issue of data and branch hazards becomes more complicated when multiple instructions are issued at once, since an instruction cannot depend on the results of any instruction issued at the same time, nor on the results of any instruction issued on the next one or more clocks. With multiple instructions issued per clock, this increases the potential for interaction between instructions, of course.

a) Example: If a CPU issues 4 instructions per clock, then up to seven instructions following a branch might be in the pipeline by the time the branch instruction finishes computing its target address. (If it is the first of a group of 4, plus a second group of 4.)

b) Example: If a CPU issues 4 instructions per clock, then there may need to be a delay of up to seven instructions before one can use the result of a load instruction, even with data forwarding as described above.

B. Dealing with Hazards on a Superscalar Machine

1. Data hazards

a) We have previously seen how data forwarding can be used to eliminate data hazards between successive instructions where one instruction uses a result computed by an immediately-preceeding one. However, if "producer" and "consumer" instruction are executed simultaneously in different execution units, forwarding no longer helps. Likewise, the unavoidable one cycle delay needed by a load could effect many successive instructions.

b) Superscalar machines typically incorporate hardware interlocks to prevent data hazards from leading to wrong results. When an instruction that will store a value into a particular register is issued, a lock bit is set for that register that is not cleared until the value is

actually stored - typically several cycles later. An instruction that uses a locked register as a data input is not issued until the register(s) it needs is/are unlocked.

c) Further refinements on this include a provision that allows the hardware to schedule instructions dynamically, so that a "later" instruction that does not depend on a currently executing instruction might be issued after an "earlier" instruction that does. (This is called OUT OF ORDER EXECUTION.) Of course, the hardware that does this must avoid reordering instructions in such a way as to change the meaning of the program [e.g. interchanging two instructions that both store a value in the same place, since the final store is the one that "sticks"]

2. Branch hazards

a) Stalling the pipeline until the outcome of a conditional branch is known is one possible solution - but it can get expensive, since quite a number of instructions could be issued in the time it takes a conditional branch instruction to get to the point where its outcome is known.

b) Another way to deal with branch hazards is to use branch prediction to speculatively issue several instructions before the outcome of a conditional branch is known.

(1) A branch history table can be be used to "remember" the target address of branch instructions to allow moving down the "branch taken" path if this is the predicted outcome. (Otherwise, the pipeline would have to stall if branch taken is predicted.)

(Since the target address of a branch instruction is generally computed by PC + displacement in instruction, a given branch instruction will always point to the same target.)

(2) If a prediction turns out to be wrong, the pipeline is flushed and quite a bit of work may have to discarded. (However, the loss is no greater than if the CPU had stalled until the branch outcome is known).

(3) In any case, though, prediction requires the ability to reach a definitive decision about whether the branch is going to be taken before any following instructions have stored any values into memory or registers.

c) An alternative to branch prediction that is used on the Intel Itanium RISC architecture (formerly IA64) is called PREDICATION. In this strategy, the CPU includes a number of one bit predicate registers that can be set by conditional instructions. The nstruction format includes a number of bits that allow execution of an instruction to be contingent on a particular predicate register being true (or false). Further, a predicated instruction can begin executing before the value of its predicate is actually known, as long as the value becomes known before the instruction needs to store its result. At that point, if the predicate is false, the storage of the instruction's result is inhibited.

This can eliminate the need for a lot of branch instructions.

Example:

```
if r10 = r11 then
        r9 = r9 + 1
else
        r9 = r9 - 1
```

Would be translated on MIPS as:

```
        bneq    $10, $11, else
        nop                 # Branch delay slot
        br      endif
        addi    $9, $9, 1   # In branch delay slot
else:
        addi    $9, $9, -1
endif:
```

Which is 5 instructions long and needs 4 clocks if $10 = $11 and 3 if not.

But on a machine with predication as:

```
set predicate register 1 true if $10 = $11
(if predicate register 1 is true) addi $9, $9, 1
(if predicate register 1 is false) addi $9, $9, -1
```

Which is 3 instructions long (all of which can be done in parallel, provided the set predicate instruction sets the predicate register earlier in its execution than the other two store their results.)

C. Advanced CPU's use both superpiplining and superscalar techniques. The benefits that can be achieved are, of course, dependent on the ability of the compiler to arrange instructions in the program so that when one instruction depends upon another it occurs enough later in the program to prevent hazards from stalling execution and wasting the speedup that could otherwise be attained.