Objectives
----------
1. To introduce the basic concept of CPU speedup through pipelining
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.

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

\[
\text{Time} = \frac{\text{cycle time} \times \# \text{ of instructions} \times \text{CPI}}{\text{clock-rate}} = \frac{\# \text{ of instructions} \times \text{CPI}}{\text{clock-rate}}
\]

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 inductance, 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 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. Note that, 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. It appears, though, that it is only possible to reduce CPI at the cost of more instructions or a slower clock.
5. 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. We now look at several speed-up techniques - of increasing sophistication - whereby the different phases of several successive instructions are done in parallel. To some extent, what we will look at is applicable to any architecture; but it becomes most fully implementable for RISCs.

1. For our examples, we will assume a CPU that executes an instruction in 5 distinct steps: fetch the instruction, decode it, and then up to 3 steps to execute it. Of course, on CISCs the number of steps per instruction varies widely. Even on a RISC, not all instructions need all the execution steps.

2. The timing of the CPU operations as we have discussed them thus far can be pictured as follows, using a Gantt chart (where S1, S2, S3 are a series of instructions being executed one after the other):

<table>
<thead>
<tr>
<th>Step 5</th>
<th>-S1-</th>
<th>-S2-</th>
<th>-S3-</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 4</td>
<td>-S1-</td>
<td>-S2-</td>
<td>-S3-</td>
</tr>
<tr>
<td>Step 3</td>
<td>-S1-</td>
<td>-S2-</td>
<td>-S3-</td>
</tr>
<tr>
<td>Decode</td>
<td>-S1-</td>
<td>-S2-</td>
<td>-S3-</td>
</tr>
<tr>
<td>Fetch</td>
<td>-S1-</td>
<td>-S2-</td>
<td>-S3-</td>
</tr>
<tr>
<td>Time</td>
<td>------</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes:

a. To avoid tying the chart to any specific implementation, generic names are used for the instruction phases after the first two. (Regardless of implementation, the first two steps in any instruction are, of necessity, fetching it and decoding it.)

b. The chart makes two simplifying assumptions that may or may not be valid for a given architecture:

   i. All instructions have the same number of steps.

   ii. Each step takes the same amount of time.

3. By building additional hardware, it is possible to speed up execution by doing portions of two or more instructions in parallel.
D. One of the simplest speedup techniques is PRE-FETCHING of instructions.

Step 5
- S1-
- S2-
- S3-

Step 4
- S1-
- S2-
- S3-

Step 3
- S1-
- S2-
- S3-

Decode
- S1-
- S2-
- S3-

Fetch
- S1-
- S2-
- S3-

Time -------->

1. Though each instruction still takes 5 cycles from start to finish, the average number of clocks per instruction - in the steady state - is four - i.e. one instruction completes every 4th cycle. This represents a 20% 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. If the outcome of a conditional branch sends the program down a different path from the one the CPU has been prefetching on, then the queue is flushed.

   d. The requirement that RISC architecture imposes that all instructions be the same length (one word) facilitates prefetching.

E. Further parallelism between instructions can be achieved at the cost of increased complexity of hardware.

1. For example, suppose that we not only prefetched instructions, but also overlapped the decoding of each instruction with the execution of its predecessor.

   Step 5
   - S1-
   - S2-
   - S3-

   Step 4
   - S1-
   - S2-
   - S3-

   Step 3
   - S1-
   - S2-
   - S3-

   Decode
   - S1-
   - S2-
   - S3-

   Fetch
   - S1-
   - S2-
   - S3-

   Time -------->

   [ In this - idealized - case we have reduced average per instruction time by 40%. ]
2. Perhaps we could go further and overlap the different stages of execution of two successive instructions. (E.g. fetch S2 at the same time S1 starts execution).

a. One complication can arise if the current instruction is a conditional branch. In this case, one cannot know while the instruction is being executed whether to prefetch the next sequential instruction or the one at the branch address. This is known as a BRANCH DEPENDENCY or BRANCH (or CONTROL) HAZARD.

b. With simple prefetching, this problem could be avoided by ensuring that a conditional branch instruction requires less than three steps to execute (so the decision and target address are known before the next instruction needs to be fetched). However, as the degree of overlap between instructions increases, it eventually will become a problem.

c. Some machines that suspended prefetching of instructions during execution of a branch instruction until the outcome is known (typically some time prior to Step 5).

d. Other machines always prefetch and perhaps start to execute the next instruction, or always prefetch and perhaps start to execute the branch target. If the guess is wrong, then another fetch of the correct instruction must occur before further computation can be done.

e. 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?

i. 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 then
  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.

ii. 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.

iii. 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.

f. Other machines attempt to prefetch both ways and then discard the wrong instruction.
g. Some machines (such as MIPS) _always_ execute the instruction after
the branch instruction, even if the branch is taken. (Note: on
MIPS, special hardware is used for calculating the branch address
in the decode step, so that a branch is actually completely
executed in just two steps.)

3. The ultimate degree of parallelism would be total overlap between all
phases of different instructions - e.g.

<table>
<thead>
<tr>
<th>Step</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>-S1--S2--S3--S4--S5--S6--S7--S8-</td>
</tr>
<tr>
<td>4</td>
<td>-S1--S2--S3--S4--S5--S6--S7--S8-</td>
</tr>
<tr>
<td>3</td>
<td>-S1--S2--S3--S4--S5--S6--S7--S8-</td>
</tr>
<tr>
<td>Decode</td>
<td>-S1--S2--S3--S4--S5--S6--S7--S8-</td>
</tr>
<tr>
<td>Fetch</td>
<td>-S1--S2--S3--S4--S5--S6--S7--S8-</td>
</tr>
</tbody>
</table>

Time -------->

[We have now reduced average per instruction time by 80% !]

a. 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 possible.

b. Of course, an average CPI of 1 is attainable only when the pipeline
is full of valid instructions. If the pipeline must be flushed,
it may take several cycles for the pipeline to fill up again.

c. This turns out to be the structure of most MIPS implementations.
(On MIPS, the decode step includes copying the operand values from
general registers into the holding registers of the ALU)

d. We have already seen how overlapping successive instructions can
lead to control hazards. When we get to full pipelining, though,
an additional complication arises.

i. Suppose 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: lw $2, some-address
S2: addi $3, $2, 1

(Clearly, the intention is for S2 to use the value stored by S1)
If S2 needs this operand on its first execution step (Step 3) and
S1 doesn't store it until its last step (Step 5), then the value
that S2 gets will be the PREVIOUS VALUE in the register - not the
one stored by S1, as intended by the programmer.

ii. This sort of situation is called a DATA HAZARD or DATA
DEPENDENCY. We will discuss how it can be handled shortly.
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.

II. A Three-Stage Pipeline

A. Rather than discussing the MIPS pipeline, which has five stages, we will
look at a simpler, three-stage pipeline, based on that of one of the
earliest RISC's: Berkeley RISC.

1. Stage one is instruction fetch that both fetches an instruction from
the memory address pointed to by the PC and updates the PC.

2. Stage two is an ALU operation
   a. Processing at this stage involves
      i. Reading two source values out of appropriate register(s)
         and/or a field in the instruction.
      ii. Performing some basic operation (e.g. add, sub, etc.) and
          storing the result.

      (Note: MIPS does each of these in a separate stage. There is no
      reason why they cannot all be done in one stage if we are willing
to accept a longer clock cycle - which is, of course, the motivation
for breaking them into two stages on MIPS!)

   b. The precise function performed depends on the instruction type
      i. For R-Type instructions, this step does the actual computation,
         and stores the result in a holding register, from which it will
         be copied to its destination on the next step.
      ii. For memory reference instructions, this step calculates the
          address, which will be used to transfer data to/from memory on
          the next step.
      iii. For branch type instructions, this step calculates the target
          address and (if appropriate) updates the PC with the new value.
          (Thus, a branch instruction is completed by the end of stage 2).
3. Stage three handles the result of the instruction. Again, the precise function depends on the instruction.

   a. For R-Type instructions, the value that was computed in the ALU in stage 2 is stored into the appropriate register in the register file.

   b. For Memory reads, a value is read from memory into the appropriate register using the address calculated by stage 2.

   c. For memory writes, a value is written to memory from the appropriate register using the address calculated by stage 2.

   d. Branch instructions do not use this stage. (It does nothing).

4. To allow these three stages to be pipelined, the CPU is organized as follows:

   ![Diagram of CPU organization]

   a. There are three functional units - an instruction fetch unit, an ALU, and a load/store unit - corresponding to the three stages of instruction execution. To facilitate this, there are two access paths to memory - one used by the instruction fetch stage (instruction memory port) and one used by the load/store stage (data memory port) - two different paths to the same memory system.

   b. As an instruction is executed, it is passed from stage to stage - like stations on an assembly line. It spends one basic clock cycle at each stage. There are two instruction registers - one between the instruction fetch stage and the ALU stage, and one between the ALU stage and the load/store stage.
i. At the end of each cycle, the instruction fetch unit puts a new instruction into the IR between the first two stages.

ii. At the end of each cycle, the instruction that is in the first IR is copied into the second one.

c. Three instructions are in the pipeline at any time - one at each stage. Thus, although it takes up to 3 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 threefold speedup.

B. Because the pipeline is so regular in its operation, the compiler can use knowledge about the pipeline to optimize the code it generates.

1. We've already noted that one problem faced by pipelined CPU's is data dependencies.

a. In this case, if one instruction loads a value into a register and the very next uses that same register as an input, then we have a problem since the value isn't actually put into the register until the clock at the end of stage 3, which is after the next instruction needs this for stage 2.

b. One approach to handle such a situation is a pipeline stall, or "bubble".

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

ii. In such cases, the hardware can replace the instruction in the first IR with a NOP and force the IF stage to refetch the same instruction instead of going on to the next.

iii. Of course, this means wasting a clock cycle, since the NOP does no useful work.

c. A better approach called DATA FORWARDING is available in some cases. 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 available in the ALU Holding Register - it just hasn't yet been placed into the correct register in the register file. In such a case, the hardware can detect this situation and forward the value directly from the holding register to the ALU input (while also storing it into the correct register on the next clock so it is available to future instructions.

However, this doesn't work for memory load instructions. Why?

ASK

The value needed hasn't yet been read from memory.

d. For such cases, we can require the compiler to anticipate such a problem by never 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 (using the assembly language notation of Berkeley RISC).

```
load r10, a ; r10 <- a
load r11, b ; r11 <- b
nop -- inserted to prevent data conflict
add r11, r10, r11 ; r11 <- r10 + r11
load r12, c ; r12 <- c
nop -- inserted to prevent data conflict
add r12, r11, r12 ; r12 <- r11 + r12
add r12, r12, #1 ; r12 <- r12 + 1
store r12, d ; d <- r12
```

However, the NOPs can be eliminated by rearranging the code:

```
load r10, a ; r10 <- a
load r11, b ; r11 <- b
load r12, c ; r12 <- c
add r11, r10, r11 ; r11 <- r10 + r11
add r12, r11, r12 ; r12 <- r11 + r12
add r12, r12, #1 ; r12 <- r12 + 1
store r12, d ; d <- r12
```

This approach - requiring that code not use a register immediately after it has been loaded - is called DELAYED LOAD. (If an instruction just after a load does try to use the same register as a source, it gets the OLD value.)

e. A final alternative is to incorporate interlocks in the hardware that actually stall the pipeline if an attempt is made to use a register that is currently being loaded.

i. This 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, MIPS implementations since MIPS ISA III have done this, though most earlier MIPS implementation required the compiler to prevent the problem.

ii. To the extent that the compiler can schedule instructions to avoid hazards, a program will run faster, since stalls will not be needed.

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

a. Branch instructions are executed in the second stage of the pipeline and the branch target address (and decision whether or not the branch is to be taken) are not available until the end of the cycle. Thus, while the branch is being executed, the next sequential instruction is being fetched by the instruction fetch unit.
b. In this case, attempting to somehow predict the outcome of the branch does not help, because the TARGET ADDRESS of the branch becomes available at the same time the outcome becomes known - if we predict the branch to be taken in advance of knowing its outcome, we cannot do anything with the prediction because we don't yet know where the branch will go (unless we used a branch history table, which Berkeley RISC doesn't).

c. Again, one way to handle this situation is with a "bubble" - if the branch is taken, then the instruction behind it is converted to a NOP.

d. The approach that was used on Berkeley RISC is the same as that used by MIPS (and a number of other RISCs) - DELAYED BRANCH.

i. All control transfer instructions (subroutine calls and returns as well as jumps) take effect AFTER the next instruction in sequence is executed.

Example: Suppose we were compiling

```plaintext
if something then
    a := a + 1;
else
    ...
```

Suppose further that a is a local variable allocated to reside in r16.

Then the code for the "then" part would consist of an add 1 to r16 plus a branch to skip over the "else" part. This could be done this way:

```plaintext
addi r16, r16, 1  r16 <- r16 + 1
jmp end_if
nop
```

else_part:

```plaintext
d... end_if:
```

The nop is needed because the instruction after the jmp is always executed - it's in the pipeline before the branch is actually done.

However, a good compiler would emit the code this way:

```plaintext
jmp end_if
add r16, r16, #1  r16 <- r16 + 1
```

else_part:

```plaintext
d... end_if:
```

ii. As the above example illustrates, the compiler can normally work with this feature of the hardware by inserting the JUMP instruction ahead of the last instruction to be done in the current block of code. In some cases, though, a NOP must be inserted (e.g. if the jump is a conditional that depends on the last operation to be done before the jump is taken.) (Recall the NOPs after all branches in the programs in lab - to avoid having to deal with this issue right away.)
III. Moving Beyond Basic Pipelining

A. 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

   \[
   \text{w/o pipeline - 1 instruction / 4 ns} \\
   \text{with pipeline - 1 instruction / 1 ns} \\
   \text{4ns/1ns = 4:1}
   \]

   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.

      i. Branch hazards can be reduced by doing the relevant computations in an early 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.

      ii. Data hazards, again, may lead to a need to use interlocking to ensure correct results.

3. 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.

B. 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. Because 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.

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 simultaneously 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.

Note: in effect, a group of instructions that do a computation - say - on an array element and then update a pointer to point to the next element become, in effect, like a single CISC instruction with autoincrement!

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 requires 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 the same architecture.

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

5. Current practice - found on a number of RISCs, including the Power PC and IA64 - 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. A superscalar CPU has just one instruction fetch unit (which 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.

C. 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 make even more extensive use of branch prediction, perhaps speculatively issuing several instructions before the outcome of a conditional branch is known.

i. 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.)

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

iii. 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 being used on the Intel Itanium RISC architecture (Itanium) is called PREDICATION. In this
strategy, the CPU includes a number of one bit predicate registers that can be set by conditional instructions. The instruction 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. 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
  br endif
  # Branch delay slot
else:
  addi $9, $9, 1
  # In branch delay slot - always done
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.)

D. 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.