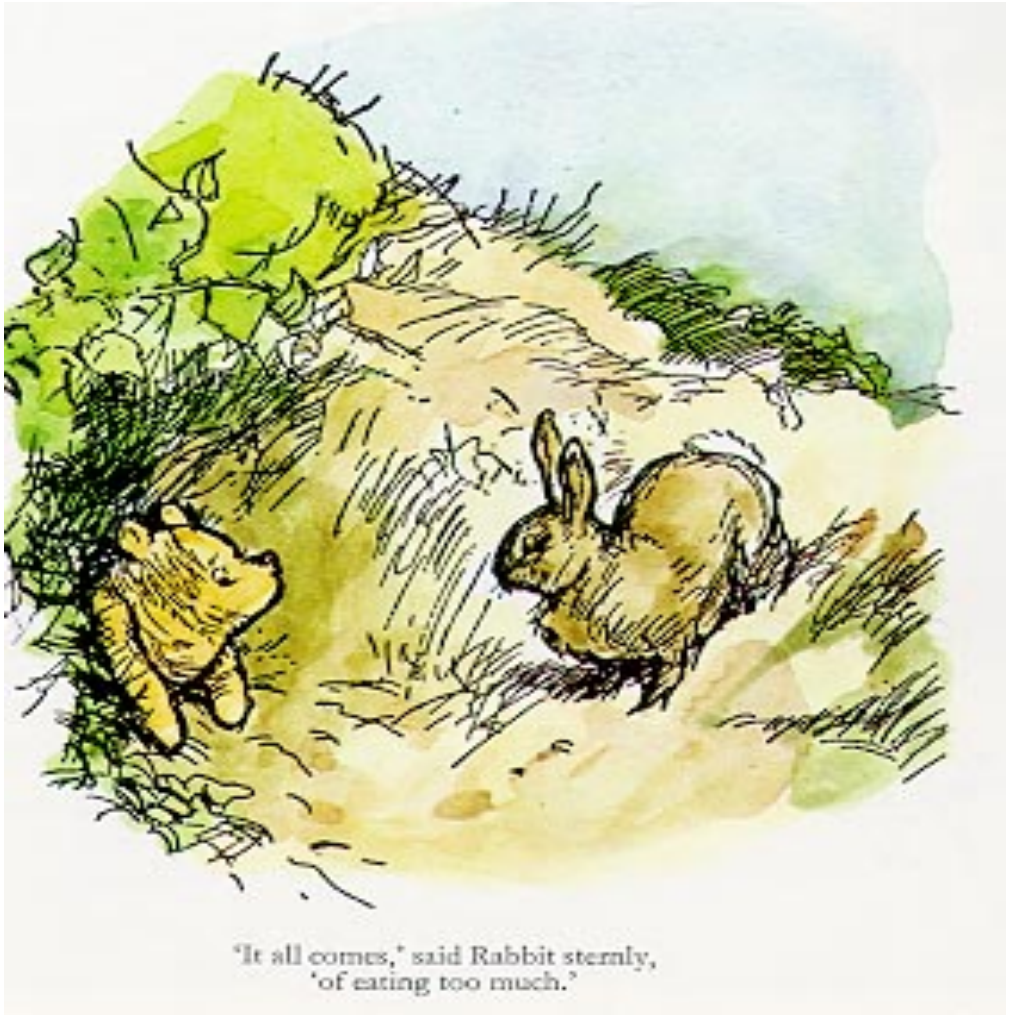
Preventing Stalls: 1



Pipeline efficiency

Pipeline CPI = Ideal pipeline CPI
+ Structural Stalls
+ Data Hazard Stalls
+ Control Stalls

Ideal pipeline CPI: best possible (1 as n $\rightarrow\infty$)

Structural hazards: Insufficient hardware

Data hazards: Need results of earlier calculations

Control hazards: Need to foretell the future.

Instruction-Level Parallelism (ILP):
Seeks to overlap instruction execution

Hardware: dynamically runtime
Software: statically compile time

Simplest is Loop Level Parallelism

Branches and jumps

3 Loops

Loop level parallelism

Dynamic: branch prediction
Static: Loop unrolling

Dependence
independent/parallel. Simultaneous execution possible. Can be placed in a pipeline, with only (possibly) structural hazards.

dependent.. Must occur in order; partial overlap possible

Dependent: if instruction2 uses result of instruction 1. *Data Hazard.*
Read after Write Hazard : RAW Hazard

Hardware and Software must produce the same result as strict sequential execution.

May not be correct, if code is not correct

Preserve order only where vital

Actual hazard: existence; stall length. Depends on implementation of pipeline.

Dependence in program
indicates *potential* for hazard.
stipulates an order
upper bound on possible performance

Pipe Effic

Name dependence

Also called anti-dependence.
When a memory location or register is re-used.

Means instruction 2 may write before instruction 1 has used the value
Write after Read, WAR hazard.

Or instruction 1 may write after instruction 2 has written, but before the subsequent use is made of the data
Write after Write, WAW hazard.

Both may be resolved by using a separate register.
register renaming (hardware or software)

Control Dependence
if … then else blocks

May not be correct, if code is not correct

Preserve order only where vital

Often blocks can be executed ignoring conditions, if we can throw away the results.
Ensure the system is completely unaffected by unwanted calculations.

Need to handle *exceptions* and ensure correct *data flow*

Assembler

```
for (int pnt=1000; pnt>0; pnt--) {
   arr[pnt] = arr[pnt] + offset;
}
```

This Java loop will compile to something like

```
lw   $r2, offset;          offset
Loop:
lw   $r3, 0($r1);          arrEl element
     add  $r4,$r3, $r2;    add
     sw   0($r1), $r4;     store result
     addi $r1, $r1, -4;    decrement pnt
     bnez  $r1, Loop       continue
```

MIPS 5 step pipeline becomes

```
lw   $r2, offset;
Loop:
     lw   $r3, 0($r1);
     stall                  waiting for $r3
     add  $r4,$r3, $r2;
     stall                  waiting for $r4
     stall
     sw   0($r1), $r4;
     addi $r1, $r1, -4;
     stall                  no forward to
bnch
     bnez  $r1, Loop;
```

May not be correct, if code is not correct

Why one stall and then two

Re-ordering

```
lw  $r2, offset;
Loop:
    lw  $r3, 0($r1);
    stall                   waiting for $r3
    add  $r4,$r3, $r2;
    stall                   waiting for $r4
    stall
    sw   0($r1), $r4;
    addi $r1, $r1, -4;
    stall                 no forward to
bnch
    bnez  $r1, Loop;
```

9 cycles

```
lw  $r2, offset;
Loop:
    lw  $r3, 0($r1);
    addi $r1, $r1, -4;    decrement early
    add  $r4,$r3, $r2;    removes stall
    stall                 waiting for $r4
    stall
    sw   8($r1), $r4;     displacement
    bnez  $r1, Loop;      $r1 already done
```

Reordered – 7 cycles

Unrolling

```
lw  $r2, offset;
Loop:
    lw  $r3, 0($r1);
    stall                          stall comes back
    add  $r4,$r3, $r2;
    stall*2
    sw   0($r1), $r4;
    lw  $r2, offset;
                                   second iteration
    sw   -4($r1), $r4;
    lw  $r2, offset;
                                   third iteration
    sw   -8($r1), $r4;
    lw  $r3, -12($r1);
    stall
    add  $r4,$r3, $r2;
    stall*2
    sw   -12($r1), $r4;
    addi $r1, $r1, -16;   decrement 4
times
    bnez  $r1, Loop;
```

Unrolled – 26 cycles 6.5 per iteration

Harder if total number is not divisible by the unroll number

General upper bound U

Loop unroll m times. Execute unrolled loop U/m and original loop U mod m.

Optimise unrolled loop

```
lw  $r2, offset;
Loop:
    lw  $r7,  0($r1);
    lw  $r8,  -4($r1);
    lw  $r9,  -8($r1);
    lw  $10, -12($r1);
    addi $r1, $r1, -16;  decrement 4
times
    add  $r3,$r7, $r2;   stall hidden
    add  $r4,$r8, $r2;
    add  $r5,$r9, $r2;
    add  $r6,$r10, $r2;
    sw   0($r1), $r3;
    sw   4($r1), $r4;
    sw   8($r1), $r5;
    sw   12($r1), $r6;
    bnez  $r1, Loop;
```

Unrolled optimised – 14 cycles 3.5 per iteration

cf 9 originally – speed up > 3

Decisions

• Unrolling useful if iterations are independent
• Use different registers to avoid name dependence. Sets limit on size of unroll
• Eliminate test and branch instructions. Will need to modify them at the end of the code
• Independent iterations allow reorder of load and store between loops
• Ensure code delivers same result.

Note number of iterations depends on length of code.

Unroll too many times for a long loop and may start generating cache missed for the code.

Also run out of independent registers.

Long loops have little overhead from house keeping code. Marginal advantage of extra iterations decreases.

Long loops have other ways to hide stalls.

Split Instruction Decode

Instruction Decode **ID**. Step of the pipelining.
Checks for structural and data hazards
Split into two

Issue: Decode instructions. Structural hazards ?
Wait for data hazards to clear
Read Operands:

Extension of 5 step pipeline to out of order execution creates possibility of WAR and WAW

Solved by register renaming.

Out of order execution creates problems for exceptions.

Imprecise exceptions raised – exceptions which do not look as if the instructions were executed sequential
• Instructions earlier than the exception may not have completed
• Instructions later than the exception may have completed

CDC 6600

CDC 6600

131 kWords

Three of the 16 page frames in the 6600.

3 million instructions per second

The word length of the 6600 central memory is 60 bits.

There are 10 built-in independent Peripheral and Control Processors in the 6600 Computer, each having a 4096-word core memory. These memories are **in addition** to the 6600 central memory.

There are 10 functional units in the 6600 Central Processor, as follows:

- 2 Adders
- 2 Multipliers
- 2 Incrementers
- 1 Divider
- 1 Shift
- 1 Boolean
- 1 Branch

These functional units operate on 8 increment registers of 18-bit length, 8 operand registers of 60-bit length, and 8 memory address registers of 18-bit length. Up to 32 instructions can be held at once for program loops.

There are several levels of concurrency in the 6600 Computer, broadly consisting of:

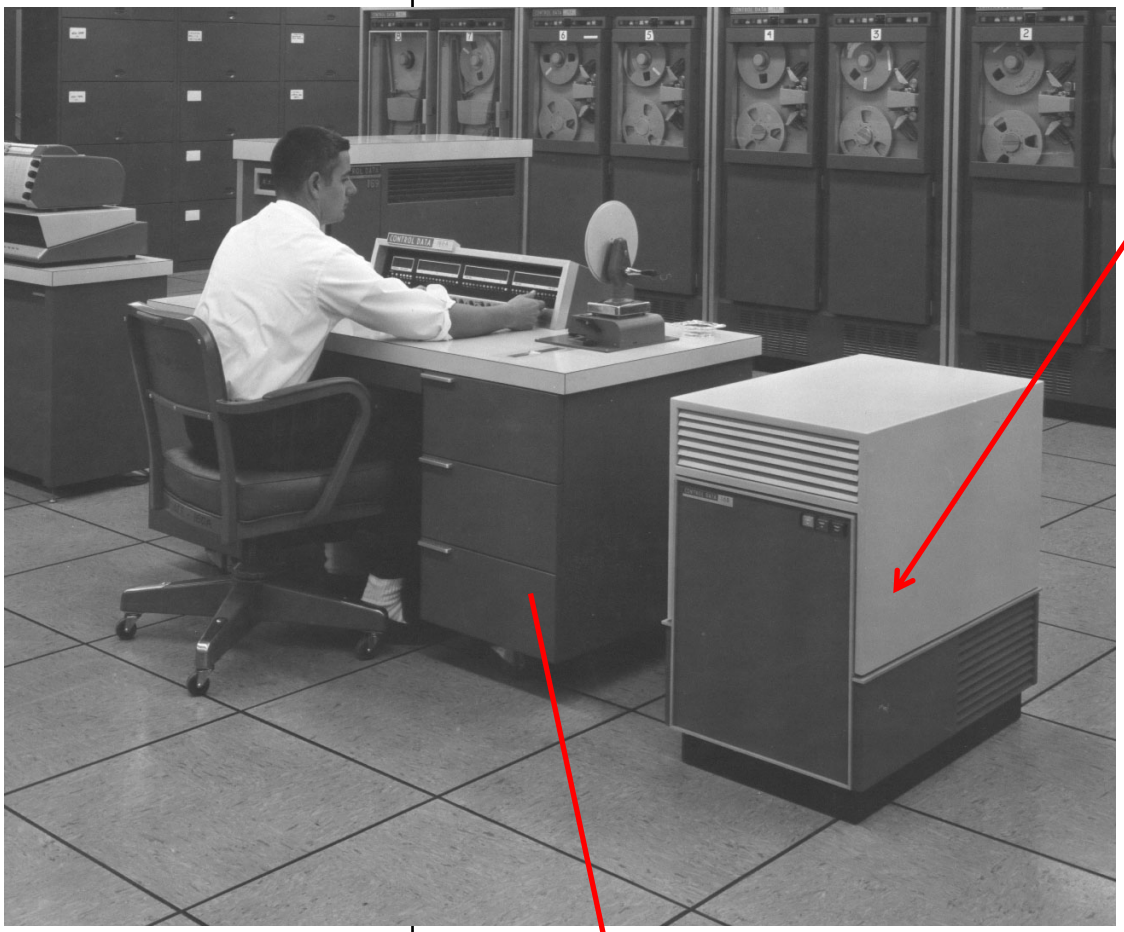
- Concurrency in program execution in the 10 built-in Peripheral and Control Processors.
- Concurrency in the 10 basic Central Processor functions.
- Concurrency in the 32 independent banks of the Central Memory.

Eleven programs are run simultaneously on the 6600 Computer. These programs are **not** time-shared.

The 6600 Computer provides maximum operator/machine communications via its console tube display, which includes keyboard.

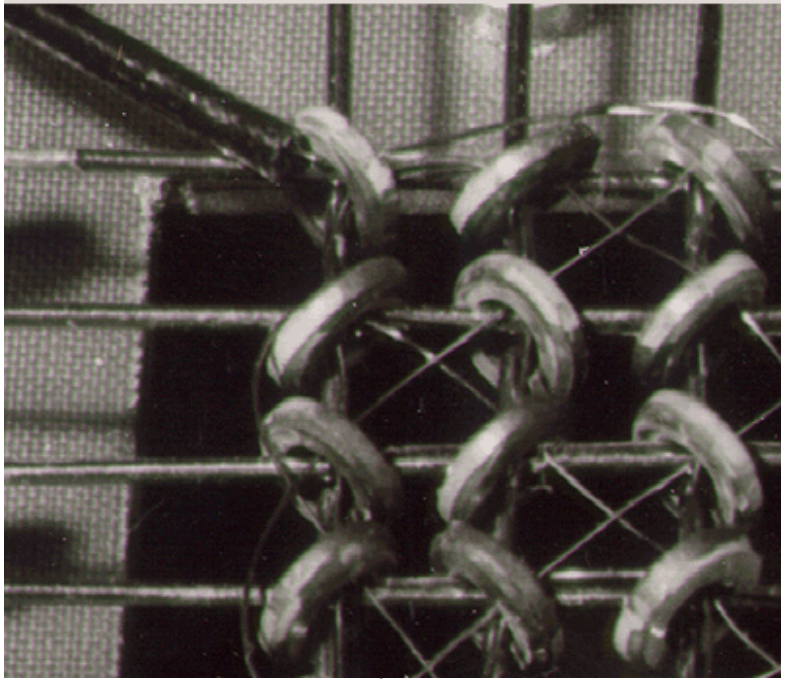
 CDC 160

Computer (and operator)



Multiply/divide unit

Computer around 1960

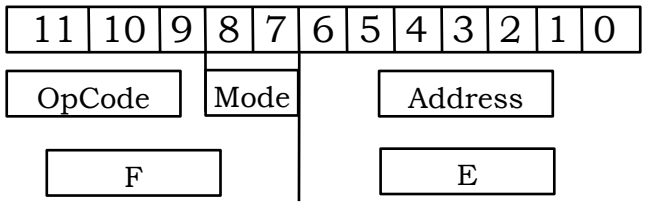


CPU and core memory

Instructions

12 bit word

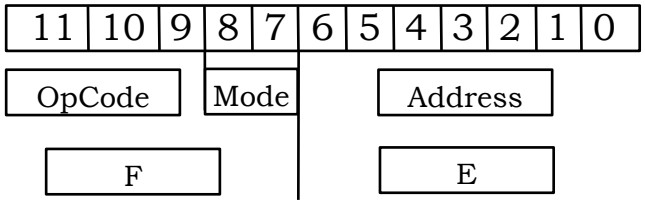
Registers PC;
Accumulator;
address register 4k words



0010 – And 0011 - Or
0100 – Load 0101 - Load Complement.
0110 – Add 0111 - Subtract.
1000 – Store 1001 - Shift and Replace.
1010 - Add and Replace.
1011 - Add One and Replace.

00 or 10

Addressing modes



Direct — Fetch contents of E (only 6 bits of address space)

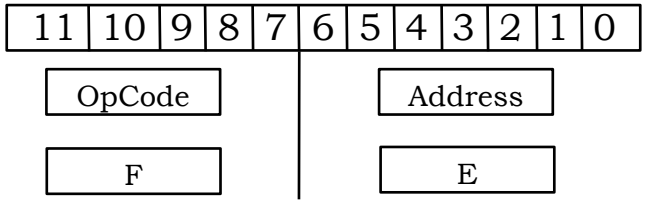
Indirect. — If E=0 next word holds address (address all memory)
If E not zero it points to a word (from 0-64) which holds the address of the next instruction

Forward Relative: Next instruction is PC+E field

Backward Relative: Next instruction is PC-E field

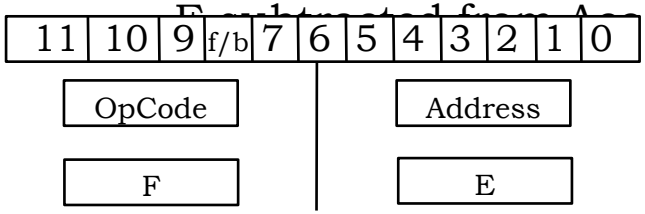
All possible modes used.
And limits instructions to 16

Immediate instructions



Small constants can be added in one instruction

000010 – And — E (zero extended) is added to Acc
000011 – Or — E or'd with Acc
000100 - Load. — E into Acc
000101 - Load Complement — E complement into ACC
000110 – Add — E added to Acc
000111 – Subtract — E subtracted from Acc



Relative jumps

110x00 - Zero Jump. — All bits 0
110x01 - Non-Zero Jump. — 1 bit non 0
110x10 - Positive Jump. — A>0
110x11 - Negative Jump. — A<0

Bit 8 jump forward or backward

Indirect Jumps
111000 - Jump Indirect.
111001 - Jump Forward Indirect.

Shifts
000001000010 - Shift Left.
000001000011 - Shift Left 2 (SN > 37).
000001001000 - Shift Left 3.
000001001001 - Shift Left 6 (SN > 37).
000001001010 - Multiply by 10.
000001001011 - Multiply by 100 (SN > 37).

Control Instructions

000000000000 - Halt.
111111111111 - Error.
000001000001 - Transmit Program Counter into Accumulator.

No specific function calls to allow storage of PC and transfer of control. TPC Allows then store and jump in fewer instructions.

Note more than 16 instructions. So instructions are effectively vartiable length

Scoreboarding

Developed for the CDC 6600 in the mid 1960's

Execution:

Goal; 1 instruction per cycle.
Instructions executed as early as possible

When no structural hazards

Instruction stall
proceed to subsequent instructions
Execute unless they depend on previous executing (or stalled) instructions.

Many instructions in simultaneous execution
Need hardware to match

CDC Load/store

CDC6600 4 FP units, 5 Memory Refs, 7 integer ops

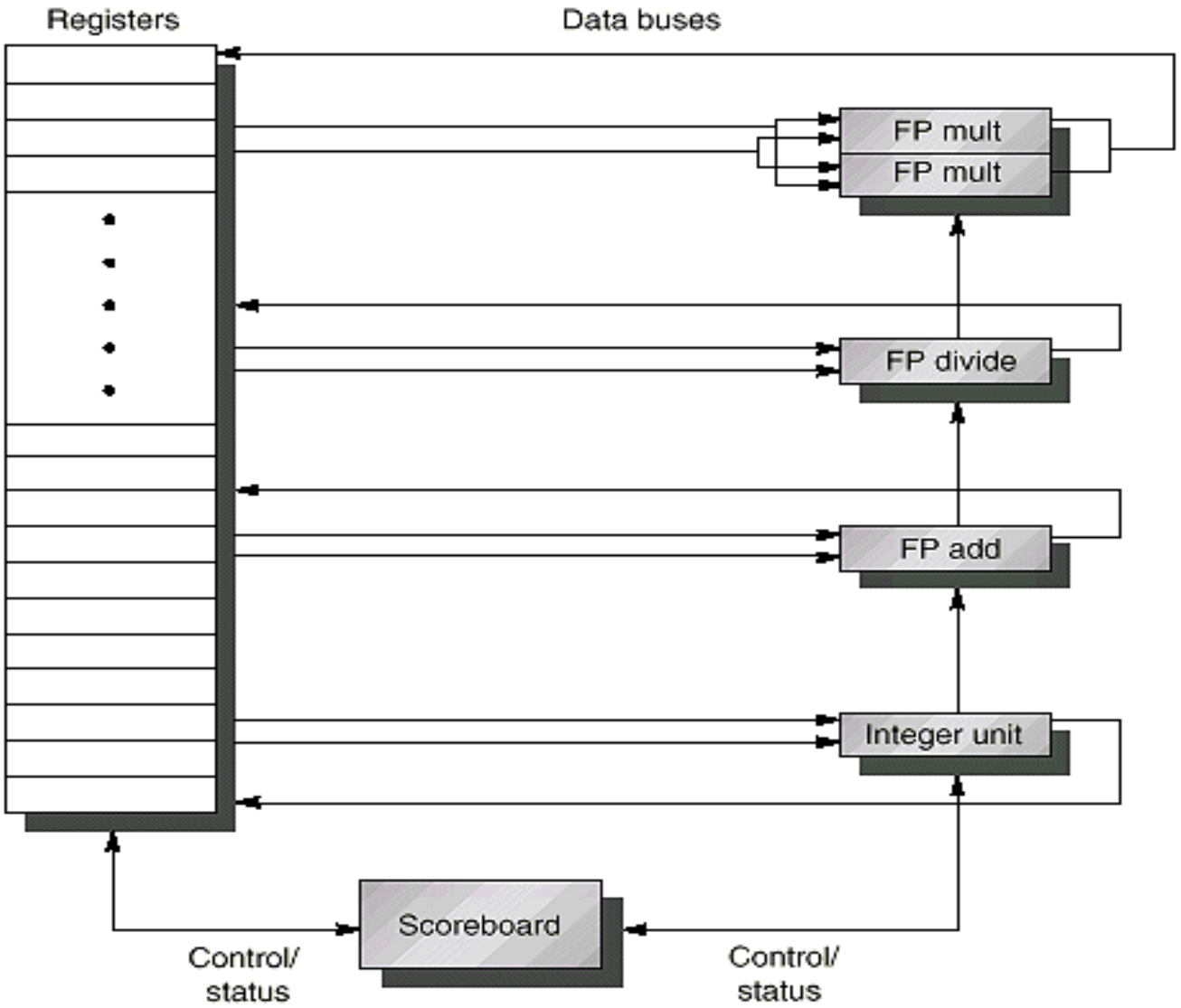
MIPS only FP units

The scoreboard handles
hazard detection
Example

See Hennessey Appendix A

Two multiplier units, one adder, one divide, one for integer adds and all memory reference calculations

Pipe Effic



See Hennessy Appendix A

Static v Dynamic scheduling

Static scheduling: can be done at compile time.

Some things are not defined at compile time.

If we have an instruction like

```
MUL             F0, F1, F2
```

The instruction cannot proceed until the operands are available. Would like to start some other instructions.

How many depends on how long?

If the contents of F1 need to be loaded from memory, then this instruction may need to wait.

For how long?

Depends if F1 is fetched from cache or memory.

That will not be known until run time.

Dynamic scheduling: provides a mechanism to keep the pipeline flowing, using information not available at compile time.

Scoreboard

Structural hazards cab be mitigated by increasing the number of functional units available (FP add, FP mult,…) and distributing the instructions between them. This takes extra logic.

The scoreboard is some extra circuitry which takes the instructions and distributes the FP operations between multiple functional units – add, multiply, divide.

The aim (as always) is 1 instruction per clock cycle

Three of the steps in the standard MIPS pipeline

ID,EX,WB

Are replaced by

Issue, Read Operands, Execution, Write Results

Issue decode instructions, check for structural hazards

Read operands wait until no data hazards, then read operands

Scoreboard

In order	*issue*
Out of order	*execution*
Out of order	*commit*

The scoreboard consists of three parts

Instruction status
Functional unit status
Register result status

Instruction status
Which step the instruction is executing

Functional unit status
The status of each functional unit

Busy	yes/no	
Op		operation … add, subtract, …
Fi		destination register
Fj, Fk	source registers	
Qj, Qk	functional units producing Fj, Fk	
Rj, Rk	yes, registers ready & not read	

Register result status

Which functional unit will write to each register
If a functional unit has this register as its destination

Issue: Functional Unit Free and no other unit has same destination register. Scoreboard issues instruction and updates its internal data structure.
Protects against WAW hazards.
If issue stalls the buffer between IF and Issue fills.

Read Operands:
Scoreboard monitors availability of source operands (data flow).
Source operand available if not earlier issued instruction is going to write to it
When source operands available to scoreboard tells the functional unit to begin execution.
Resolves RAW hazards – instructions may be sent to execution out of order

Execution: Functional Unit executes instruction and informs scoreboard when complete.

Write Result: Scoreboard checks for WAR hazards and stalls completing instruction if required.

Instructions may complete out of order.
Instructions may even "overtake" each other.

Execution "as if" serial

Amount of parallelism
Each instruction depends on predecessor None

Number of scoreboard entries (window size)
Determines how far ahead the pipeline can look for instructions

Number and type of functional units
How many instructions can occur in parallel. Tend to provide more multiply units, since multiply takes longer than add.

Presence of anti-dependences and dependences
Lead to RAW and WAW stalls

Balance

VAX 8650 had a cycle time of 55ns with a sophisticated pipeline.
VAX 8700 had a simpler pipeline which allowed a speed of 45ns.
8650 had 20% less CPI, 8700 was 20% faster.
But 8700 was simpler – less hardware.

Performance measurement

Compiler optimisation covers some of the same ground as dynamic scheduling.

Manufacturers ingenuity replaces simple clock speed.
Doesn't always work

Measure improvement with unoptimised code will give an over optimistic idea of improvement
Be sure what it is you are measuring

Complex system – non-linear interactions
Measurements are hard