Name Solution

Computer Architecture
EE 4720
Final Examination

18 May 2002, 12:30–14:30 CDT

Problem 1		(20 pts)
Problem 2		(37 pts)
Problem 3		(20 pts)
Problem 4		(23 pts)
Exam Total		(100 pts)

Alias Map or BAT?

Good Luck!

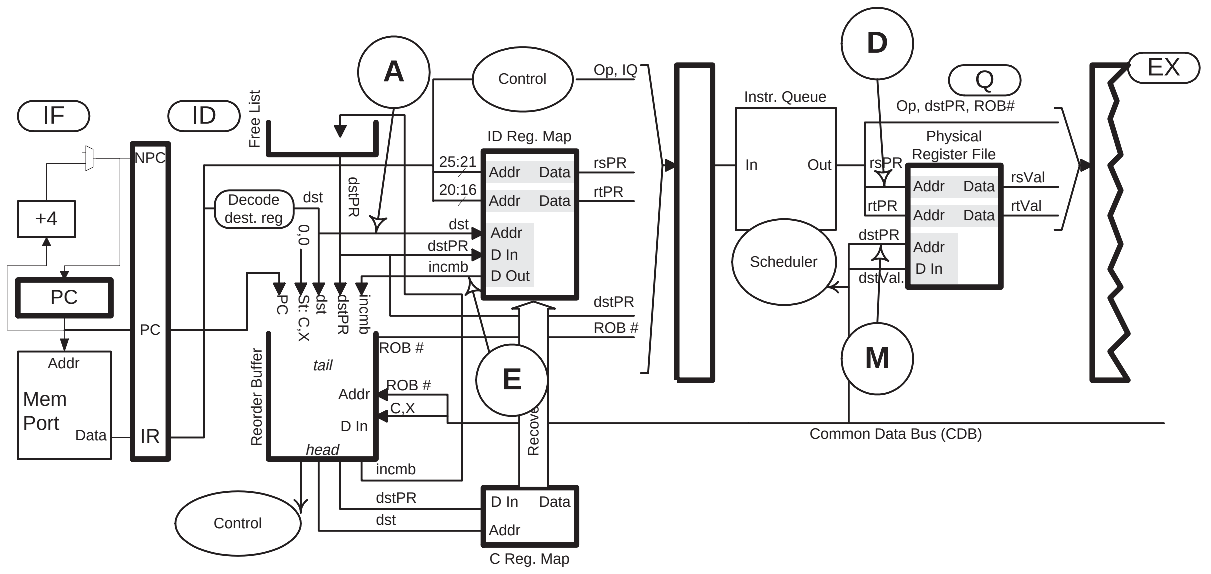
Problem 1: The (hopefully familiar) code fragment on the next page executes as shown on the dynamically scheduled system illustrated below.

(*a*) For each entry in the Physical Register File table on the next page place a "[" at the cycle(s) at which a physical register is allocated (removed from the free list) and place a "]" at the cycle(s) at which it is placed back in the free list. (This was part of the solution to Homework 5.) (10 pts)

(*b*) The diagram below includes circled letters, these letters appear in a Signals Values Table on the next page.

☑ (10 pts) Fill in the table showing the signals that will appear on the labeled wires.

- Use register names (`f0`, `$1`, etc.) for architected registers (but not for physical registers!).
- Use a question mark for a physical register number that cannot be determined from the tables.
- Since the machine is four-way superscalar each wire holds four values. Do not show values for the instructions that are not in the table, such as the two instructions following `sub`.
- Assume that values have been correctly computed, even if they depend upon preceding instructions in the same group. (This affects row E in the table.)



The filled-in tables appear on the next page.

The D row shows the physical register used for the rs operand being read for instructions starting execution. Instructions start execution when their operands are ready, with one exception that is the cycle after the instruction is decoded. The exception is multiply which waits two cycles.

The M row shows the physical register number for instructions being written back. The easy way to fill this row is to check when entries are written in the physical register file table.

```
LOOP: # Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
 ldc1 f0, 0($1)   IF ID Q  L1 L2 WC
                           IF ID Q  L1 L2 WB             C
                                    IF ID Q  L1 L2 WB             C
                                             IF ID Q  L1 L2 WB
 mul.d f0, f0, f2 IF ID Q        M1 M2 M3 M4 M5 M6 WC
                           IF ID Q        M1 M2 M3 M4 M5 M6 WC
                                    IF ID Q        M1 M2 M3 M4 M5 M6 WC
                                             IF ID Q        M1 M2 M3 M4 M5
 sdc1 0($1), f0   IF ID Q  L1                      L2 WC
                           IF ID Q  L1                      L2 WC
                                    IF ID Q  L1                      L2 WC
 addi $1, $1, 8   IF ID Q  EX WB                      C
                           IF ID Q  EX WB                      C
                                    IF ID Q  EX WB                      C
 bne $2, $0 LOOP     IF ID Q  B  WB                   C
                              IF ID Q  B  WB                   C
                                       IF ID Q     B  WB                C
 sub $2, $1, $3      IF ID Q  EX WB                   C
                              IF ID Q  EX WB                   C
                                       IF ID Q  EX WB                   C
 # ID Map         0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
 f0 99               97,96    94,93    91,90
 $1 98               95       92       89
 # In cycle one first 97 is assigned to f0, then 96 (replacing 97).  The
 # same sort of replacement occurs in cycles 4 and 7.
 # Commit Map     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
 f0 99                           97                96    94 93    91 90
 $1 98                                                95       92       89

 # Cycle          0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
 Physical Register File
 99               1.0            ]
 98               0x1000                              ]
 97                 [            10                ]
 96                 [                              11    ]
 95                 [        0x1008                            ]
 94                            [           20                ]
 93                            [                             2.2   ]
 92                            [        0x1010                            ]
 # Cycle          0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17

 # Signal Values  0    1    2    3    4    5    6    7    8    9    10   11   12
 # Solution
 A                f0   $2             f0   $2        f0   $2        f0   $2
                  f0                  f0             f0             f0
                  $1                  $1             $1             $1

 D                          98   ?    97   95   ?    94   92 ...
                            98   95        95   92        92 ...
                            98             95             92 ...


 E                     99   ?         96   ?         93
                       97             94             91
                       98             95             92

 M                                    95   97        92   94             96

 # Cycle          0    1    2    3    4    5    6    7    8    9    10   11   12
```

Problem 2: The `add` instruction below is a predicated version of a MIPS instruction (for this exam). If the contents of register `$s5` (for the example) is non-zero the `add` instruction executes normally. If it's zero then it's as though the `add` never executed. Any GPR can be used to hold the predicate.

```
 # Cycle                     0  1  2  3  4  5  6
           xor $v1, $a0, $a1  IF ID EX ME WB
           lw $s5, 0($t2)        IF ID EX ME WB
     ($s5) add $s1, $s2, $s3        IF ID EX ME WB
```

(*a*) Suppose there are predicated versions of other two-source, one-destination instructions. How might they be coded in MIPS? The coding should be chosen to ease implementation. (5 pts)

Use the Type-R format and use the SA field to hold the predicate register number.

(*b*) Modify the pipeline below to implement predication; it should run the code above correctly and without stalls (as illustrated). The added hardware must detect whether the predicate is true or false and take appropriate action. It should include bypassing, for example, to handle the dependency carried by `$s5` or from `xor` if the predicate were `$v1`.

Hint: A correct solution uses multiplexors, an $\boxed{\textit{is P}}$ *(box that recognizes a predicated instruction), assorted gates, and some modifications to existing components.*

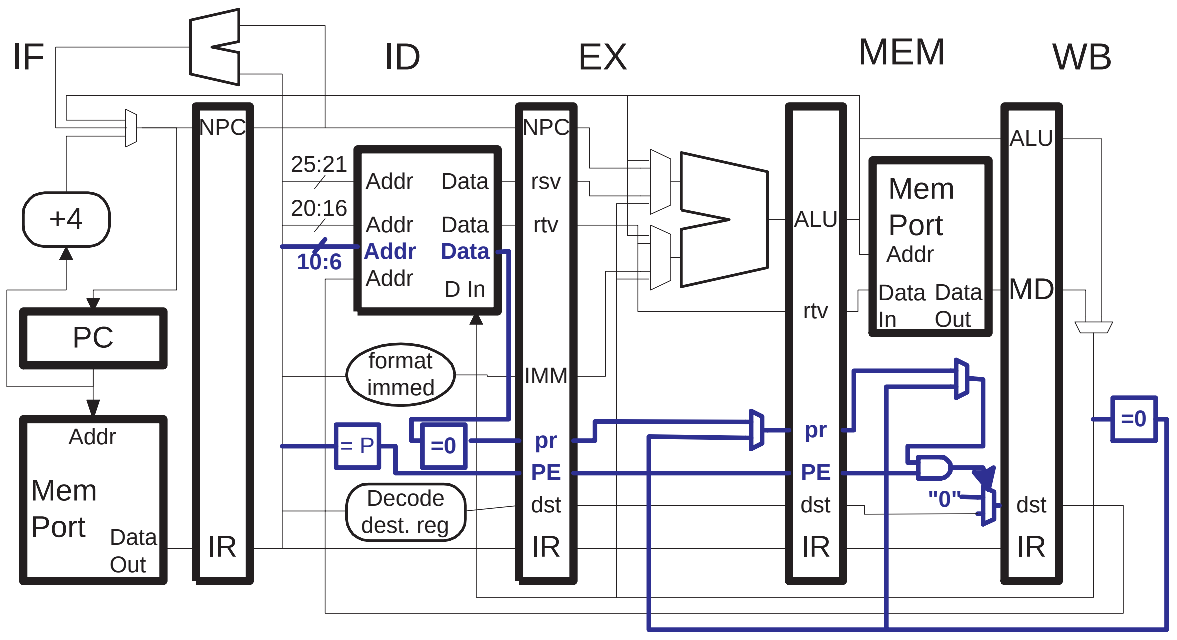
(9 pts)

Changes shown in **blue bold**.

In the ID stage a port has been added to the register file to read the predicate value, it feeds a NOR gate (to test if it's equal to zero) and is written to a new `pr` pipeline latch. The predicate is examined in the MEM stage, it its true (`pr` is 1) and the instruction is predicated (PE is 1) the "0" is select from the `dst` multiplexor, changing the destination register to zero. It's necessary to put this hardware in the MEM stage so that predicated instructions with dependencies like `add` can execute without stalling.

Bypass multiplexors are included in the EX and MEM stages, the EX mux would be used if the predicate were `v1`, the MEM mux is used, to bypass `s5`.

Note that the `pr` pipeline latches are just one bit. An alternative would be to store a full 32-bit value and check if it's zero in the MEM stage, but that would be more costly.



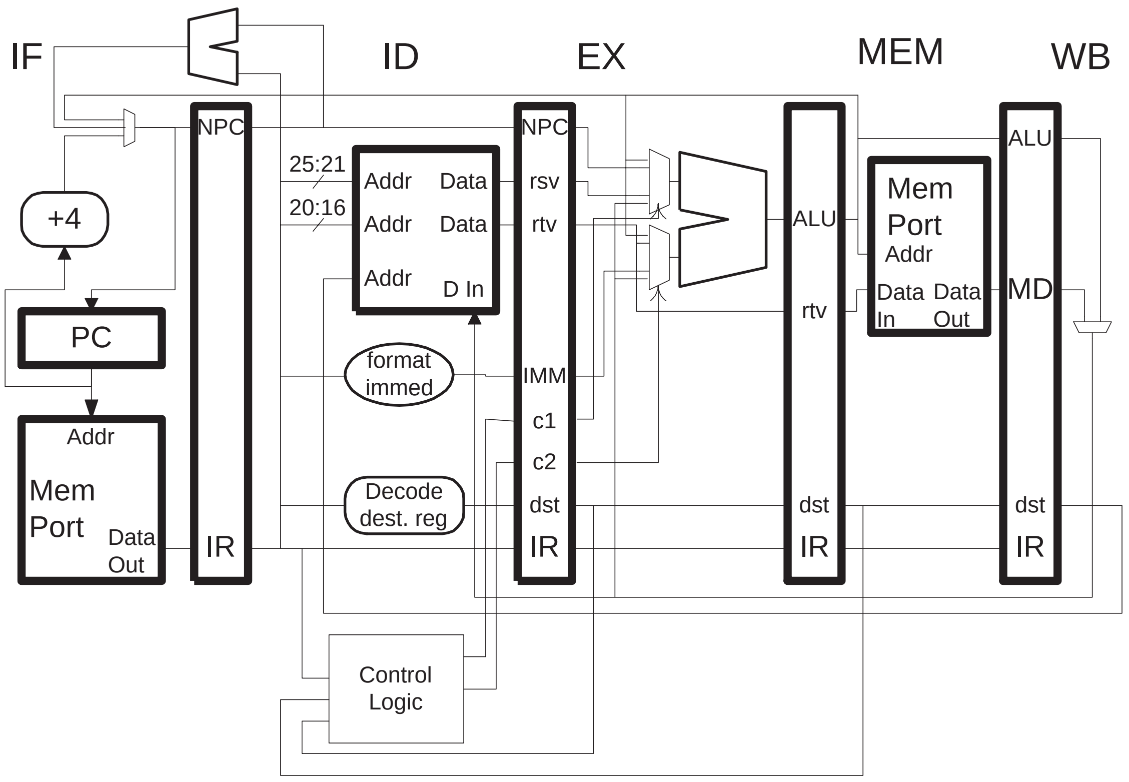
Problem 2, continued:

(*c*) Assume predicates were implemented as requested in the previous part. Explain why bypassing of the dependency carried by `$s1` in the program below won't work for the hardware below. Explain how the problem might be fixed. (8 pts)

It won't work because the bypass control logic is in the ID stage. The `or` will need to bypass a value from `add` only if its predicate is true, otherwise it will read `s1` from the register file. When the `or` is in ID the predicate on `add` is not yet resolved and so there is no way the bypass logic, which is in ID, can set the ALU multiplexors correctly.

A solution would be to move at least some of the control logic to the EX stage, and have that logic compute bypass connections based on how predicates are resolved. A less expensive and lower performance solution would be to stall instructions in ID if they depend on a value produced by an unresolved predicated instruction in EX or ME.

```
 # Cycle                     0  1  2  3  4  5  6
          xor $s5, $a0, $a1  IF ID EX ME WB
          lw $s5, 0($t2)        IF ID EX ME WB
    ($s5) add $s1, $s2, $s3        IF ID EX ME WB
          or  $s4, $s1, $v0           IF ID EX ME WB
```



Problem 2, continued: Consider such predicated instructions in a dynamically scheduled implementation using method 3. The implementation without predicate prediction (the next two parts) should realize the benefit of predicated instructions: avoiding the need for branches.

(*d*) How might the ID Register Map be updated for predicated instructions? (For partial credit: Why is this a good question?)(5 pts)

Update the ID map for a predicated instruction in the same way as an ordinary instruction: write the physical register number assigned to the destination. If the predicate turns out to be false just copy the value from the incumbent physical register to the one assigned to the instruction.

(*e*) Based on your answer above, how is execution affected if the predicate turns out to be false? (5 pts)

The only impact on execution is the need to copy a physical register and the ultimately unnecessary waiting of instructions dependent on the destination register.

(*f*) Consider a dynamically scheduled system using predicate prediction. What should be done when a predicate is mispredicted? Under what circumstances (properties of program being run) would it be faster than a system without predicate prediction? Under what circumstances would it be slower? (5 pts)

First things first. There is a big difference between a false predicate and a mispredicted one. When a predicate is mispredicted following instructions may read the wrong data. For example, if a predicate is mispredicted true then some instructions may read the value written by the predicated instruction or if its mispredicted false they read an "outdated" value rather then the one the predicated instruction would have written. In either case they read the wrong value and so will have to be squashed or re-executed. The problem is that following instructions started executing based on a predicted predicate value, not the real one. If there is no predicate prediction following instructions will wait for the predicate to resolve so they will always execute with the correct data, whether or not the predicate is true or false.

Now back to the solution: What one does on a predicate misprediction depends on how the ID map was updated. If the ID map was updated as described above then on a misprediction instructions dependent on the result can be re-executed (not something covered in class), a simpler solution would be to squash all instructions following the predicated instruction, restore the ID map and re-start fetching after the predicated instruction. If instructions are re-executed the ID map does not have to be restored because it is correct, only the values in the physical registers are wrong.

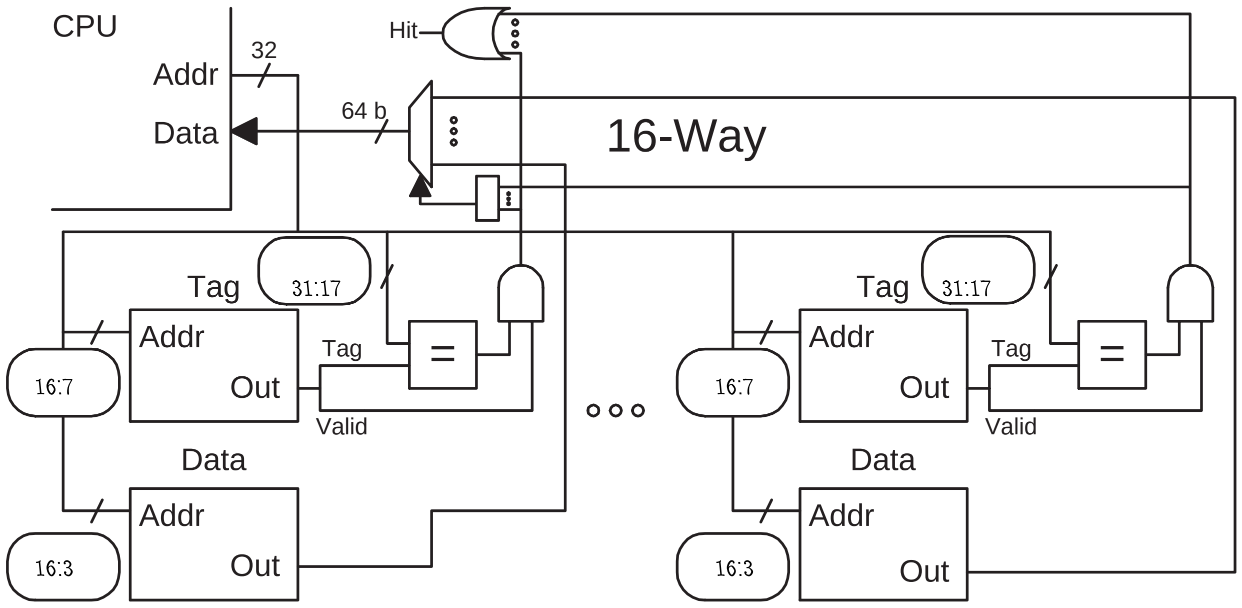
Another way to handle the ID map is to not update it for predicates that are predicted false. In that case squashing would be necessary (re-execution would not be possible) since the ID map would be invalid.

Programs benefit from predicate prediction if predicates become available later than other operands to predicated instructions and there are close dependencies with predicated instructions. It would be slower if prediction accuracy were low since valid instructions would be squashed and re-executed. There would be little benefit if predicates were never available later than other operands to the instruction. (The little benefit would be from scheduling latency, something not covered in class very much.)

Problem 3: The diagram below is for an 2 MiB (2^{21} byte) 16-way set-associative cache, with a line size of 128 characters, for a system with 8-bit (how ordinary) characters.

(*a*) Answer the following, formulæ are fine as long as they consist of literals and grade-time constants. (10 pts)

☑ Fill in the blanks in the diagram.



☑ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

	Tag	Index	Offset	
Address:	31 – 17	16 – 7	6 – 3	2 – 0

☑ Memory Needed to Implement (Indicate Unit!):

It's 2^{21} characters (data) plus $(32 - 17 + 1)$ 16 $2^{17-7} = 2^{18}$ bits for the tag store.

☑ Show the bit categorization for a direct mapped cache with the same capacity and line size.

Since it's 16-way set associative the index will need to be made $\log_2 16$ bits larger. Since the line size can't change the offset bits can't be expanded (or shrunk) and so the tag size is reduced.

	Tag	Index	Offset	
Address:	31 – 21	20 – 7	6 – 3	2 – 0

Problem 3, continued: Continue to use the set-associative cache from the previous part.

When the code below starts running the cache is empty. Consider only accesses to the array.

```
short *h = 0x100000;  // sizeof(short) = 2 characters
int sum;
int i,j;
int ISTRIDE = 1;
int ILIMIT = 1024;

for(j=0; j<4; j++)
  for(i=0; i<ILIMIT; i++)
    sum += h[ ISTRIDE * i ];
```

(*b*) Answer the following. (10 pts)

☑ What is the hit ratio for the program above?

Each line holds $\frac{2^7}{2} = 64$ short integers. Since data is read sequentially on the first `j` iteration there will be a miss followed by 63 hits for a hit ratio of $\frac{63}{64}$. The total data read in a single `j` iteration is $1024 \quad 2$ characters, which is much less than 2 MiB and so on subsequent `j` iterations all accesses will hit.

The total hit ratio is thus $\boxed{\frac{63+3 \quad 64}{4 \quad 64} = \frac{255}{256} = 0.996094}$.

☑ Find the *minimum* value of `ISTRIDE` needed to maximize the miss ratio. (That is, *minimize* the hit ratio, make things really slow.) Don't forget that the cache is set associative. Do not modify `ILIMIT`.

On a *single* `j` iteration the hit ratio can be maximized by accessing a different line every `i` iteration, that is done by setting `ISTRIDE` to one half the line size, (2^6), (since each element is two characters). *But* every access would be a hit on the next `j` iteration. *So* we need to make sure that a line cached in one `j` iteration is evicted by the next `j` iteration. Since its a 16-way cache a line is evicted after 16 different lines having the same index are accessed (assuming LRU replacement). Making `ISTRIDE` really big and a power of two would do this since then every access would have the same index. *But* the problem asked for the minimum `ISTRIDE` so we need to think a little more.

Think of `i` as a 10-bit number (since it iterates from 0 to $2^{10} - 1$). In order to get exactly 32 distinct tags for each index we need to "shift" `i` so that its five most significant bits are in the tag region, and the rest are in the index region. This way, for each index there will be 32 tags, forcing evictions. (If there were just four bits in the tag region then the cache would be able to hold the first *j* iteration.) To do that, set `ISTRIDE` to 2^{11}, and so the address will be computed by shifting `i` 12 places to the left (one extra place since a short is two characters).

So the answer is $\boxed{\text{ISTRIDE}=2^{11}}$.

Problem 4: Answer each question below.

(*a*) In the pipeline execution diagram below the multiply is squashed to avoid the WAW hazard. How does this make a precise exception impossible? (5 pts)

```
mul.d  f0, f2, f4  IF ID M1x
add.d  f0, f6, f8     IF ID A1 A2 A3 A4 WB
```

Suppose the `add.d` raises an exception. The exception handler will not see the value of `f0` written by `mul.d` (because it was squashed). For an exception to be precise the handler must see the effect of all instructions that precede the faulting instruction.

(*b*) How do processes share memory in a virtual memory system? Provide an example in which two processes share address `0x12000` but address `0x34000`, used by each process, is not shared. The addresses must be used in the example. (5 pts)

They share memory by having virtual addresses map to the same physical address. Call the two processes A and B. The page tables for process A and B might both map `0x12000` to physical page `0x1000`, and so reads and writes would be writing the same memory. Process A's page table might map `0x34000` to physical address `0x2000` while process B's page table might map it to `0x3000` and so virtual address `0x34000` would be separate for each process.

(*c*) Show how the branch history table and pattern history tables are connected in a local history branch predictor. Show how the table is indexed and how its contents are used to make a prediction. (5 pts)(For partial credit show a one-level [bimodal] predictor.)

The address input of the BHT is connected to the low bits of the program counter (or maybe NPC). The output of the BHT contains the local history for the branch, it is connected to the address input of the PHT. The PHT output is a two-bit counter, if the upper bit is 1 predict taken. A correct answer should include a sketch.

(*d*) One way to improve performance is to divide the pipeline into more stages, as in the Pentium 4 compared to the Pentium III. This does not reduce the amount of time it takes to compute things, such as sums, though. Given that: (8 pts)

☑ How are dependencies potential performance limiters when dividing pipeline stages?

The more pipeline stages are divided the higher the latency for functional units, such as the integer unit. In the classical case the integer functional unit has zero latency so dependent instructions can start in the next cycle. If it's divided in two the latency becomes one and so dependent instructions that are decoded in the next cycle will have to stall. At some point dividing pipeline stage will yield no additional performance because every instruction would be stalling.

☑ How does the Pentium 4 Fast ALU get around that?

By dividing the ALU in to low and high halves and allowing data from the output of the low half to be bypassed to the input of the low half. The same is done for the high half. Therefore even though it takes two cycles to compute a complete result (three counting conditions) a bypassable value is available in one cycle.

☑ Are dependencies still a problem or can we now use zillion-stage pipelines (at least as far as dependencies are concerned)? Explain.

Yes. This trick only works for certain integer operations, so other instructions would dominate performance. Even without those other operations, the number of bits in an integer operation is a limit.

☑ Why is branch prediction accuracy more important when there are more stages?

Because the number of instructions that are fetched between the time a prediction is made and the time it resolves is higher. For a given branch prediction accuracy more pipeline stages mean more instructions squashed. This doesn't mean lower performance, it means that performance gained by dividing pipeline stages is limited. If each pipeline stage were divided in two clock frequency and so performance, could be nearly doubled. (Nearly because of greater logic complexity, register setup times, etc.) But because roughly the same amount of time is needed to resolve a mispredicted branch (regardless the number of pipeline stages) at best the part of execution which does not include resolving mispredicted branches will nearly double.