Appendix A

Hints and Solutions

A1.1. The representation of information

Exercise 3.1. (Integer representation) (page 52)

The program:

```
#include <stdio.h>
int main (){
    unsigned char val2 = 0, val1;
    while(1){
        val1 = val2;
        val2++;
        if (val1 > val2){
printf("val1 (max. val.) = %d, val2 = %d\n",val1,val2);
            break;
            }
    }
    return 0;
}
```

produces

```
val1 (max. val.) = 255, val2 = 0
```

The number is coded in 8 bits and represents values between 0 and 255. With an integer declared as `short`, we would have:

`val1 (max. val.) = 32767, val2 = -32768`, which corresponds to two's complement, 16-bit coding.

Exercise 3.2. (Single precision floating-point representation) (page 52)

The representation $\texttt{\{00 00 72 64\}}_{10} = \texttt{\{0000 0000 | 0000 0000 | }\underline{\texttt{0}}\texttt{100 1000 | }\underline{\underline{\texttt{01000000}}}\texttt{\}}_2$. The sign bit (double underlined) is equal to `0` ("+"), the exponent (underlined) to $\texttt{\{1000 0000\}}_2$ (i.e. 128_{10}), hence the actual exponent $= 128 - 127 = 1$. The significand is $\texttt{\{1001 0...\}}_2$, i.e. $\texttt{1.10010...}_2 = 1.5625_{10}$, hence the result $2^1 \times 1.5625 = 3.125$.

Exercise 3.3. (Fixed-point and floating-point representations) (page 52)

1) - Sign: `1`;

- Integer part: `11`;

- Fractional part: `1 || 101 0001 1110 1011 1000 0 || 110 1...` (the `||` indicates the period).

```
11.1101 0001 1110 1011 1000 0110 =
1.11101 0001 1110 1011 1000 0110 2**1
```

which gives us, after rounding:

```
1   110 0000 (1...) --> E0 --> E1
0   111 1010        --> 7A
0 | 111 0100        --> 74
1 | 100 0000        --> C0
```

2) We multiply by $2^{12} = 4{,}096$, i.e. $x \times 4{,}096 = -15{,}647$ (after rounding). We convert $15{,}647$ to base 2: $\texttt{0011 1101 0001 1111}_2$. We then change the sign. We get, noting the position of the decimal point:

$\texttt{1100 . 0010 1110 0001}_2 = \texttt{C.2E1}_{16}$.

Exercise 3.4. (Short floating-point representation) (page 52)

"Implied most-significant non-sign bit" (IMSNB) should be understood as the first bit (least significant) providing an indication of the sign. Example: 32_{10} is written $\ldots 00100000_2$. The 0 bit that precedes the 1 provides an indication of the sign ($0 \rightarrow +$) and is the IMSNB. Another example: $-16_{10} = \ldots 111011111_2$ and the 1 that precedes the 0 provides the sign ($1 \rightarrow -$) and is the IMSNB.

1) Dynamic (note that the number of significant digits is on the order of 4):

i) The largest positive number is represented by $x = 01.11\ldots1 \times 2^7$:

`0111`	`0`	`1111 1111 111`

the sign and significand of which are read as: `0 | 1.1111 1111 11`.

The largest positive number is given by: $x = (2 - 2^{-11}) \times 2^7 = 2.5594 \times 10^2$.

ii) The smallest positive number is represented by $x = 01.00\ldots00 \times 2^{-7}$ since the exponent -8 is reserved:

1000	0	1000 0000 000

the sign and the significand of which are read as:

`0 | 1.0000 0000 00`, i.e. 1×2^{-7}.

The smallest positive number is given by:

$x = 1 \times 2^{-7} = 7.8125 \times 10^{-3}$.

iii) Smallest negative number: we are faced with the problem of representing negative fractional numbers in two's complement. We begin with the significand. Consider the representation $10.f_2$ with $f = 00\ldots0$. We know this is a negative number. The positive value corresponding to this number is what needs to be added to it to get 0 in the format `xx.x...` x, i.e. $10.00\ldots0_2 = 2_{10}$. Therefore, $x = 10.00\ldots0_2 = -2_{10}$.

Switching representations between a positive number x and a negative number $\tilde{x}$ can be done by noting that $x + \tilde{x} = 2^2 = 4$. This amounts to taking the complement of the representation and adding 0.00000000001.

$\lvert x\rvert$	Representation	Representation	$-\lvert x\rvert$
ϕ	ϕ	`1 0.0000 0000 000`	-2
$2 - 2^{-11}$	`0 1.1111 1111 11`	`1 0.0000 0000 001`	$-2 + 2^{-11}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$1 + 2^{-11}$	`0 1.0000 0000 001`	`1 0.1111 1111 111`	$-1 - 2^{-11}$
1	`0 1.0000 0000 000`	ϕ	ϕ

The representation of the smallest negative number is given by:

1000	1	1000 0000 000

i.e. $x = -2 \times 2^7 = -2.5600 \times 10^2$.

iv) Largest negative number: based on the above table, we get $x = (-1 - 2^{11}) \times 2^{-7} \approx -7.8163 \times 10^{-3}$.

1001	1	0 1111 1111 11

2) Positive number:

$$
\begin{aligned}
10^{-2}{}_{10} : 0.01 \times 2 &= 0.02 \\
0.02 \times 2 &= 0.04 \\
0.04 \times 2 &= 0.08 \\
0.08 \times 2 &= 1.16 \\
0.16 \times 2 &= 0.32 \\
0.32 \times 2 &= 0.64 \\
0.64 \times 2 &= 1.28 \\
0.28 \times 2 &= 0.56 \\
0.56 \times 2 &= 1.12 \\
0.12 \times 2 &= 0.24 \\
0.24 \times 2 &= 0.48 \\
0.48 \times 2 &= 0.96 \\
0.96 \times 2 &= 1.92 \\
0.92 \times 2 &= 1.84
\end{aligned}
$$

i.e. `0.0001 0010 1000 11` $= 01.0010100011 \times 2^{-4}$.

`1100`	`0`	`1 0010 1000 11`

3) Negative number: we start with the fractional part:

$$
\begin{aligned}
0.3_{10} : 0.3 \times 2 &= 0.6 \\
0.6 \times 2 &= 1.2 \\
0.2 \times 2 &= 0.4 \\
0.4 \times 2 &= 0.8 \\
0.8 \times 2 &= 1.6 \\
0.6 \times 2 &= 1.2
\end{aligned}
$$

i.e. 64.3 = `100 0000.01001 1001 1001 ...` $= 01.0000000100 \times 2^6$ or $= 01.0000000101 \times 2^6$ after rounding. The representation -64.3 can be obtained directly:

`0110`	`1`	`0 1111 1110 11`

A1.2. The processor

Exercise 5.1. (Instruction sequencing) (page 101)

Access to the second instruction byte (immediate value access) is done in M2. The content of the program counter is used before it is incremented. In cycle M3, the memory write operation uses the pair “HL” as the address.

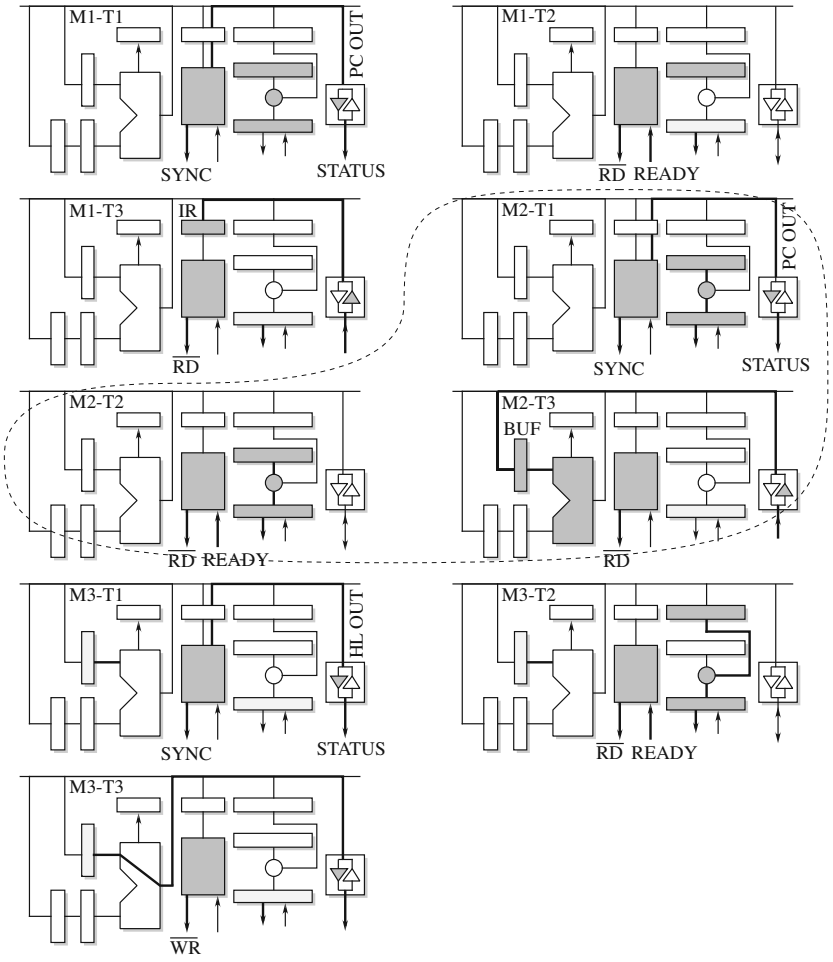


Figure A1.1. *Execution of an instruction*

A1.3. Inputs and outputs

Exercise 6.1. (Serial input) (page 133)

Based on the explanations of Figure 6.16, the read function is written as:

```
; Read function, byte in ah
loop1: mov al,[addrIN]   ; wait
       and al,1          ; test setting to zero
       jnz loop1
```

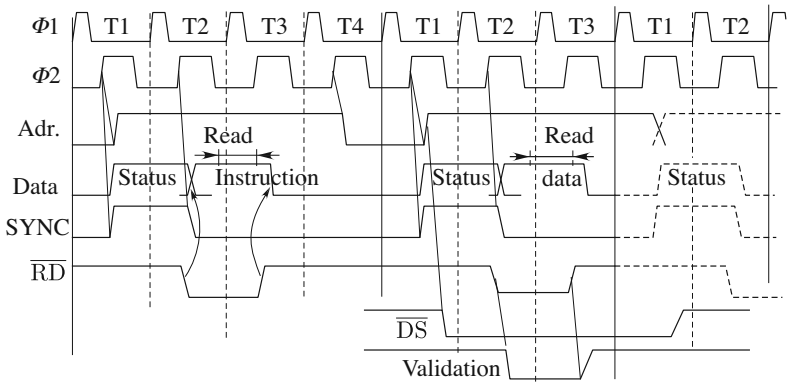


Figure A1.2. *Timing diagram for the read operation*

```
        call waitTs2      ; wait T/2 for check
        mov al,[addrIN]
        and al,1          ;
        jnz loop1
        mov cx,8
acql:   call waitT        ; wait T for next
        mov al,[addrIN]
        rcr al,1          ; store in ah
        rcr ah,1
        loop acql
        ret
```

The setting of the line to zero is tested, and after $T/2$, a check is performed. If it is still 0, the 8 bits of the character are acquired every T. A complete function would take into account all of the possible parameters, parity, number of data bits, number of stop bits, etc.

Exercise 6.2. (Serial output) (page 134)

```
; Write function of the content of ah
wfunc: xor al,al          ; al:=0
       mov [addrOUT],al   ; stop bit
       mov cx,8
wrlp:  call waitT         ; wait
       rcr ah,1
       rcl al,1           ; bit --> D0
       mov [addrOUT],al   ; write in flip-flop D
       loop wrlp
       ret
```

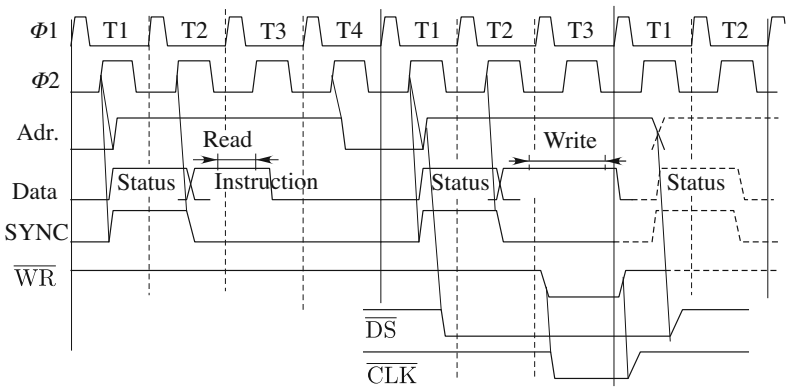


Figure A1.3. *Timing diagram for the write operation*

Figure A1.4(a) and (b) show one possible design for the input and output modules, respectively.

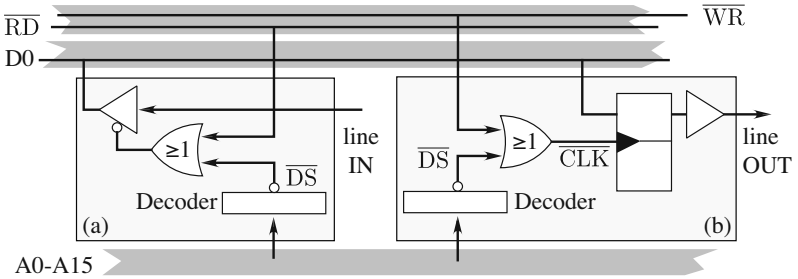


Figure A1.4. *Design diagram*

A1.4. Virtual memory

Exercise 9.1. (Virtual memory of the Am29000) (page 197)

1) General questions:

i) Operational diagram for converting "from virtual address to physical address": see Figure 9.2.

ii) When the pages are present in memory and referenced in the TLB, accessing physical addresses becomes much faster.

iii) See Chapter 6.

2) MMU parameters:

i) The supervisor mode is a "privileged" mode of operation, in which the processor has access to the entire instruction set, in particular the instructions for accessing the TLB. For security reasons, not every user is authorized to access the TLB.

ii) The *tag field* receives the VPN that will be compared to the tag field of the virtual address sent by the processor. The *Valid* bit is used to ensure a minimum level of coherence between the TLB and the content of the memory, as in a standard cache.

iii) For 1 kB pages, the maximum number of addressable pages is $2^{n-10} = 2^{22} = 4$ Mpages. The physical number of the page is 22 bits in length.

iv) This TLB is a set-associative cache with two blocks. Each line is 64 bits in length.

v) See section III-2 in the documentation: the U is used for handling the age on each line with an LRU algorithm. Depending on its value, the update will be applied to block 0 (TLB Set 0) or block 1 (TLB Set 1).

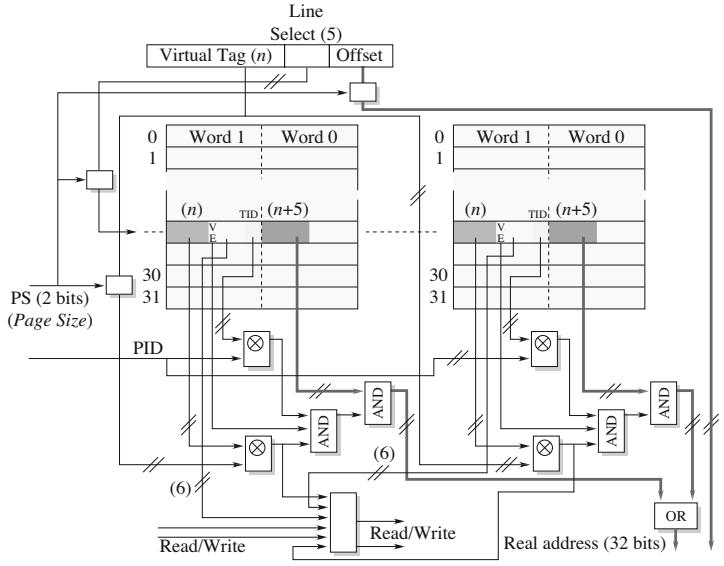


Figure A1.5. *Operation diagram of the TLB in the AMD29000*

vi) The index field still contains 5 bits. The virtual tag number is 14, 15, 16 or 17 bits in length, while the physical page number is 19, 20, 21 or 22 bits.

The TID field indicates the number of the process that owns the page. With the PID of the process currently being executed, it can be used to manage protected page access.

3) Design diagram (Figure A1.5).

A1.5. Pipeline architectures

Exercise 11.1. (Rescheduling) (page 252)

1) Without rescheduling, there are 12 instructions, and therefore 16 cycles, at best.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
lw r1,b[r0]	F	D	E	M	[W]												
lw r2,c[r0]		F	D	E	M	[W]											
add r3,r1,r2			F	D	.	[D]	E	M	[W]								
sw a[r0],r3				F	.	.	D	D	[D]	E	M	W					
lw r4,f[r0]							F	.	.	D	E	M	[W]				
lw r3,a[r0]										F	D	E	M	[W]			
sub r5,r3,r4											F	D	.	[D]	E	M	[W]

	18	19	20	21	22	23	24	25	26	27	28
sub r5,r3,r4	D	.	[D]	E	M	[W]					
sw d[r0],r5	F	.	.	D	.	[D]	E	M	W		
lw r6,g[r0]			F	.	.	D	E	M	[W]		
lw r7,h[r0]						F	D	E	M	[W]	
add r8,r7,r6							F	D	.	[D]	E	M	[W]
sw e[r0],r8								F	.	.	D	.	[D]	E	M	W

We have a total of 28 cycles, hence a penalty of 12 cycles (12 instructions, therefore 16 cycles at best: (12 "F") + "DEMW").

2) With rescheduling (Figure A1.6):

We have a total of 17 cycles, hence a penalty of one cycle compared to the unoptimized program.

Exercise 11.2. (Branches) (page 253)

1) Without forwarding (Figure A1.7):

Number of loops: $400/4 = 100$. 17 cycles per loop $\Rightarrow$ 1,700 cycles in all.

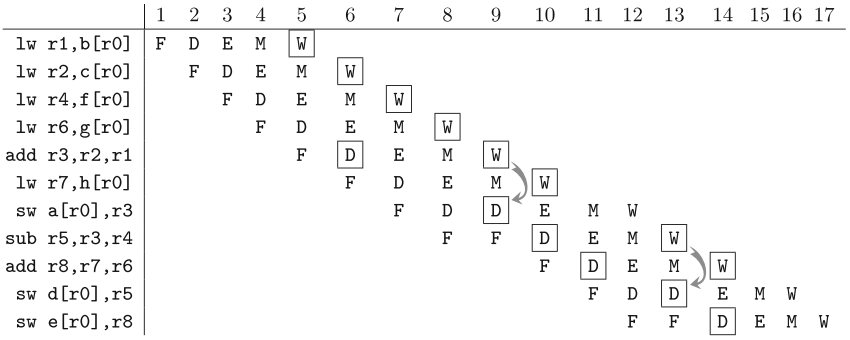


Figure A1.6. *Instruction rescheduling*

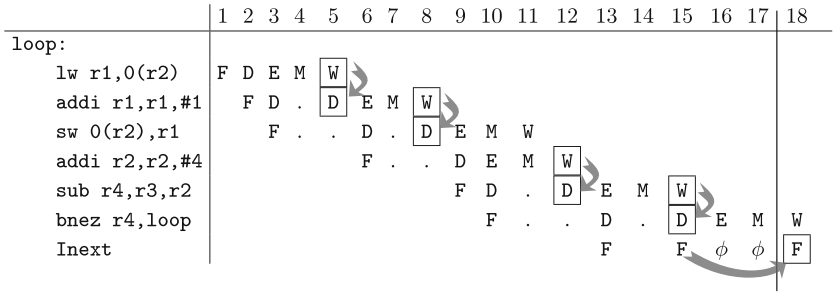


Figure A1.7. *Loop without forwarding: ϕ indicates the delays incurred because of the branch execution. Until the target address has been calculated and loaded into the program counter, there are two hold cycles. The total number of cycles is 17. On the last line,* F *denotes the fetching of the* `lw` *instruction*

2) With basic forwarding (Figure A1.8), we have a total of $100 \times 10 = 1{,}000$ cycles:

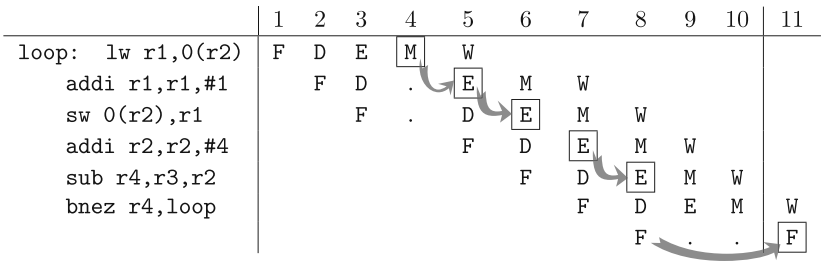


Figure A1.8. *Basic forwarding: the arrows indicate dependencies resolved using forwarding and the delay incurred because of the branch. There are 10 cycles per iteration*

3) With branch forwarding, the instruction following the branch is systematically executed. To illustrate, we move the `sw` instruction to immediately after the `bnezd`.

```
                          1  2  3  4  5  6  7 | 8
loop1:  lw r1,0(r2)      [F] D  E  M  W
   addi r1,r1,#1          F  D  .  E  M  W
   addi r2,r2,#4             F  .  D  E  M  W
   sub r4,r3,r2                 F  D  E  M  W
   bnezd r4,loop1                  F  D  E  M  W
   sw 0(r2),r1                        F  D  E  M  W
```

We get $100 \times 7 = 700$ cycles.

Instead of keeping the `lw r1` in the loop, we could insert it once before the loop and once after the `bnezd`. We must, of course, make sure that these alterations do not modify the general behavior of the program.

Exercise 11.3. (Forwarding) (page 254)

– Forwarding from EX2 to $EX1_3$:

```
   add r1,r2,r3 | F1 F2  D  EX1 [EX2]  M1   M2  WB
   sub r4,r5,r6 |    F1 F2  D   EX1   EX2  M1  M2 WB
  bnez r1,nexti |       F1  F2   D   [EX1] EX2 M1 M2 WB
```

– Forwarding from M2 to $EX1_5$:

```
    lw r1,0(r2) | F1 F2  D  EX1 EX2  M1 [M2]  WB
   sub r4,r5,r6 |    F1 F2  D  EX1 EX2  M1   M2   WB
            ... |
            ... |
 bnez r1,nexti  |              F1  F2   D  [EX1] EX2 M1 M2 WB
```

– Forwarding from M1(WB) to M1:

```
   add r1,r2,r3 | F1 F2  D  EX1 EX2 [M1(WB)]  M2  WB
    sw 0(r4),r1 |    F1 F2  D  EX1   EX2    [M1] M2 WB
```

Exercise 11.4. (Cache operation) (page 255)

The cache is a two-way set associative with one instruction per line. The four least significant bits select a fully associative, two-line cache, and the age is given by the LRU bit.

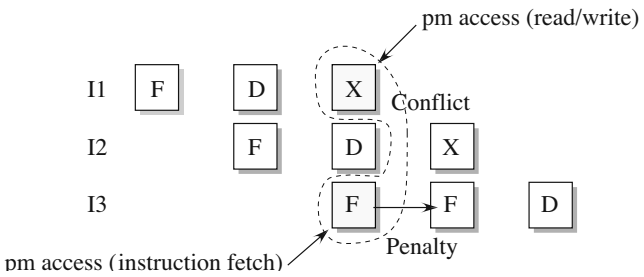


Figure A1.9. *Pipeline and cache*

1) Pipeline operation diagram (Figure A1.9): if `I1` accesses `pm` during the execution phase (Figure A1.9), it will be in conflict with the `I3` instruction, which is in phase `F`.

2) Only certain instructions are placed in the cache: those located at the address $n + 2$ in the event of a conflict with the current instruction with the address n, as indicated in Figure A1.9.

3) *Cache is enabled* indicates that the cache is operational for reading and writing. In the *frozen* mode, only read operations occur in the cache, and all updates are forbidden.

4) Cache structure:

- diagram of the cache (Figure A1.10);
- *Valid Bit*: coherence management;
- *LRU bit*: age management.

5) Consider the program in a loop situation. The first program memory access (instruction at 101) causes the instruction in 103 (in this case the conditional branch) to be stored in the cache with an index equal to 3.

When we enter the subroutine, the first access to 201 likewise causes the instruction in 203 to be stored in the cache with an index equal to 3. The instruction in 211 triggers the same event with the same index. The branch instruction is then overwritten by the instruction in 213. After the execution of the subroutine, moving on to instruction 101 leads to another miss, etc.

In all, we have a total of two *misses* for each execution of the loop. The cache operation is inefficient for this particular sequence (which has, however, a low probability of occurring).

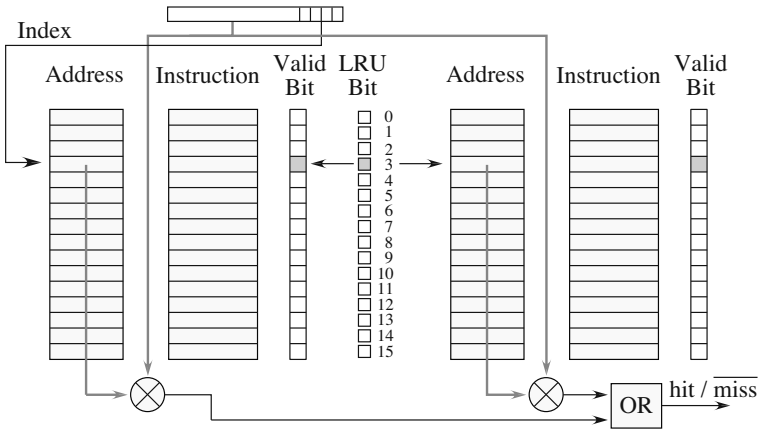


Figure A1.10. *The cache in the ADSP 2106x*

Exercise 11.5. (Branch forwarding) (page 258)

```
                sgt r4,r1,r2
                beqz r4,label1
                nop
                add r4,r2,r0        →       sgt r4,r1,r2
                j label2                    beqz r4,label1
                nop                         add r3,r1,r0
label1:         add r3,r1,r0                add r4,r2,r0
label2:                             label1:
```

Exercise 11.6. (Loop with annul bit) (page 258)

```
                lw r5,nmax(r0)
                add r4,r0,r0
                j label2                    lw r5,nmax(r0)
                nop                         j,a label2
label1:         J1                  label1: J2
                ...                 →       ...
                Jn                          Jn
                add r4,r4,#1                add r4,r4,#1
label2:         slt r6,r4,r5        label2: slt r6,r4,r5
                beqz r6,label1              beqz,na r6,label1
                nop                         J1
```

We gain one instruction per iteration.

Exercise 11.7. (Executing a subroutine call instruction) (page 259)

As in the case of branches, we lose three cycles (Figure A1.11).

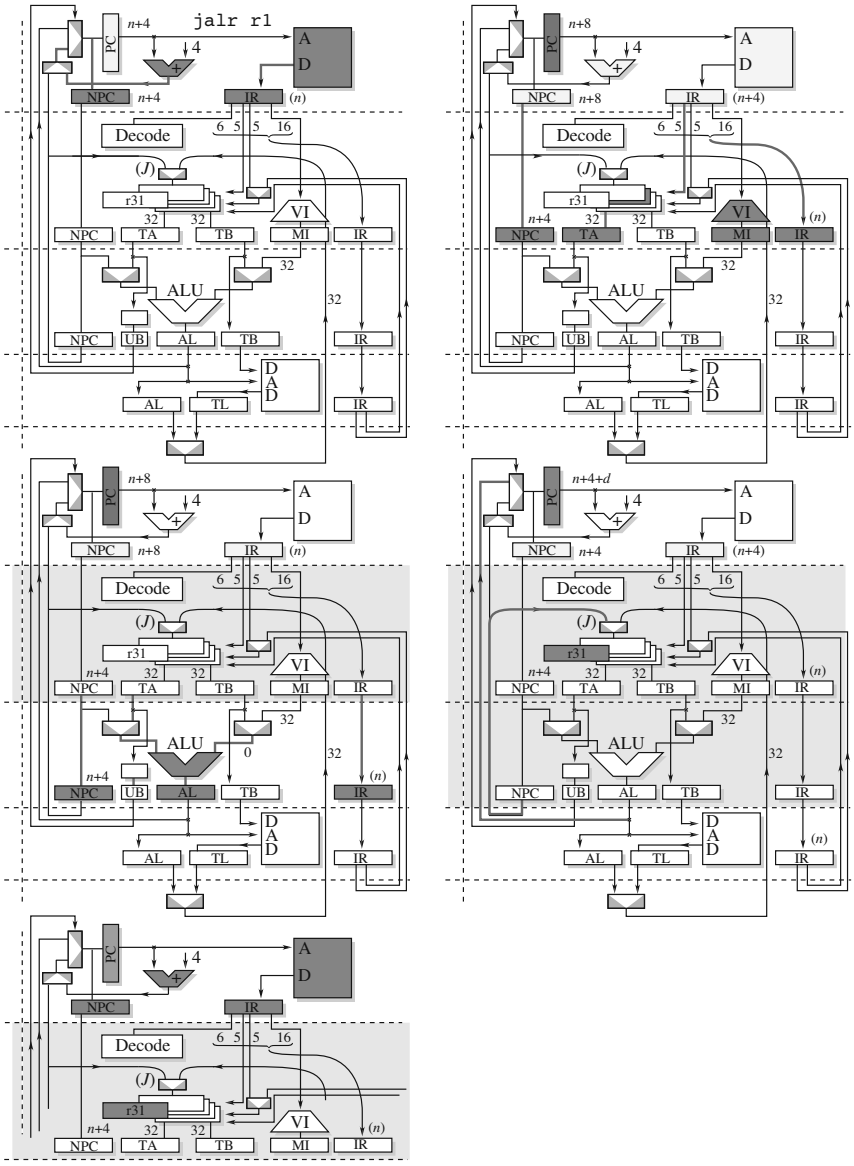


Figure A1.11. *Execution of the* `jalr`

A1.6. Caches in a multiprocessor environment

Exercise 12.1. (MSI protocol) (page 284)

Running the program leads to the exchanges shown in Figure A1.12.

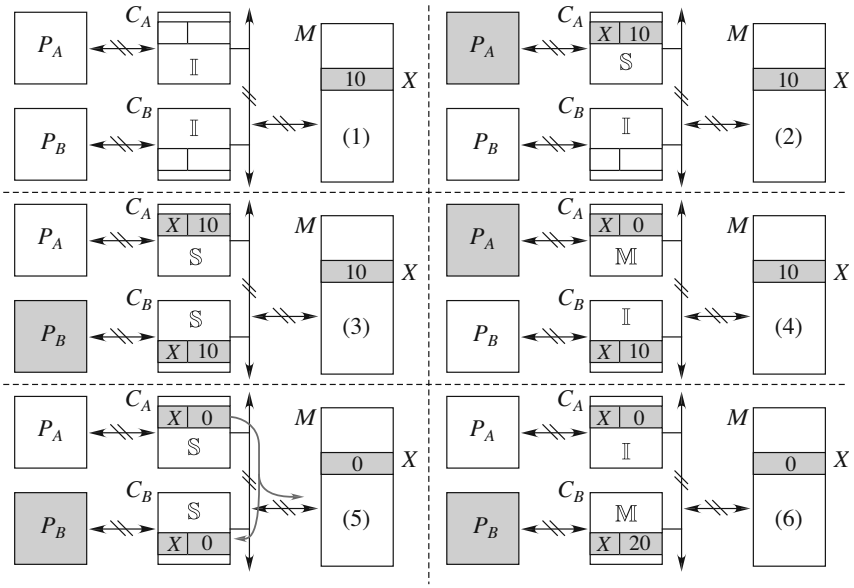


Figure A1.12. *Description of the MSI protocol operation*

REMARK A1.1.–

1) Initial situation.

2) Variable X is read by P_A: cache C_A is updated with the value 10. The line switches to the $\mathbb{S}$ state in C_A.

3) Variable X is read by P_B: cache C_B is updated with the value 10. The line switches to the $\mathbb{S}$ state in C_B.

4) P_A writes 0 in its cache. The line switches to the $\mathbb{M}$ state. The lines in the other caches switch to $\mathbb{I}$.

5) P_B wants to read X. P_A responds to the SR signal by writing the value it is storing in memory and switches to the $\mathbb{S}$ state. Cache C_B is updated and the line switches to $\mathbb{S}$.

6) P_B writes 20 in its cache. The line switches to the $\mathbb{M}$ state. C_A invalidates the line.

Exercise 12.2. (MSI protocol, update sequences) (page 284)

The value of the index is 8 for each access (`[X]`, `[X+2]`, `[X+6]`). The line is therefore "shared". Figure A1.13 describes the execution of the protocol.

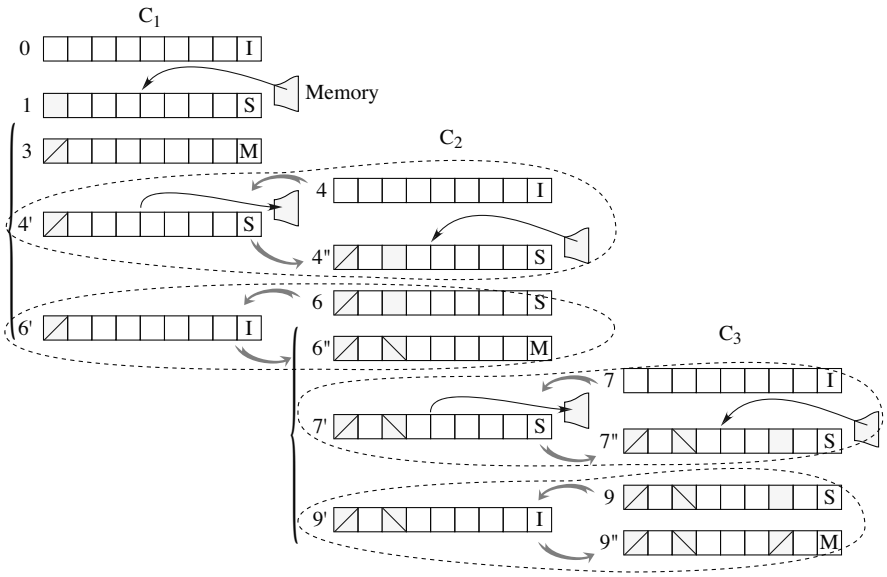


Figure A1.13. *Description of the protocol operation*

In step 4, the read request from P_2 triggers a memory update by P_1, with a switch to the $\mathbb{S}$ state for the corresponding line ($4'$). The memory transfer to C_2 is then performed, with a switch to the $\mathbb{S}$ state ($4''$).

Exercise 12.3. (MESI protocol) (page 285)

n			C_A	C_B	Memory
1	$R_A(X)10$	Miss	10-E		10
2	$R_B(X)10$	Miss	10-S	10-S	10
3	$W_A(X)0$	Invalidation in C_B	0-M	I	10
4	$R_B(X)0$	Coherence	0-S	0-S	0
5	$W_B(X)20$	Invalidation in C_A	I	20-M	0

Exercise 12.4. (Berkeley protocol) (page 285)

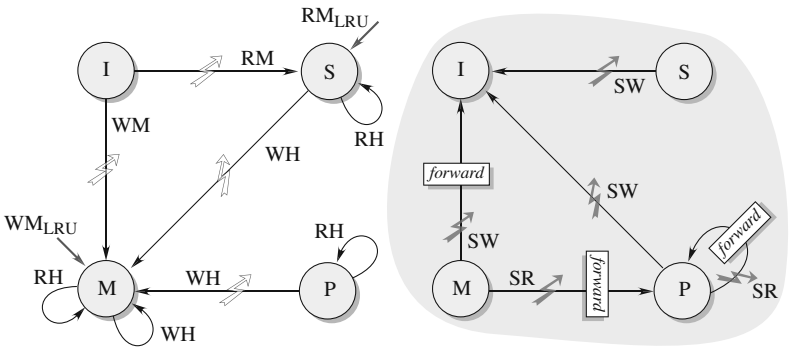


Figure A1.14. *Berkeley protocol: the processors P_0 and P_i*

REMARK A1.2.–

– Read Hit (RH): there is no communication on the bus. Accesses are performed without delays.

– Write Hit (WH): if the line is in the $\mathbb{M}$ state, access is performed with no exchange on the bus. If the line is in the $\mathbb{P}$ or $\mathbb{S}$ state, any possible copies are invalidated and the line switches to the $\mathbb{M}$ state.

– Read Miss (RM) and Write Miss (WM): see explanations on the *Read Miss* LRU and the *Write Miss* LRU.

Exercise 12.5. (Firefly protocol) (page 286)

A description of the protocol is shown in Figure A1.15.

– For an RH, there is no change of state.

– For an RM, the P_i detect and signal possession of the information if applicable (by setting the *SharedLine*). In that case, either they all have the same data, since coherence is maintained in the $\mathbb{E}$ and $\mathbb{S}$ states, or the line is in the $\mathbb{M}$ state and it is first stored into memory. In any case, if the line is held somewhere else, it can be directly loaded from cache to cache instead of being read in memory.

If no other cache contains the data, a read operation is performed in memory and the state switches to $\mathbb{E}$.

– For a WH, if the line is in the $\mathbb{M}$ state, the only write operation is in the cache (*write-back*, *w-b*). If the line is in the $\mathbb{E}$ state, it is updated and switches to the $\mathbb{M}$ state. If the line is in the $\mathbb{S}$ state, write operations are performed in the cache and the

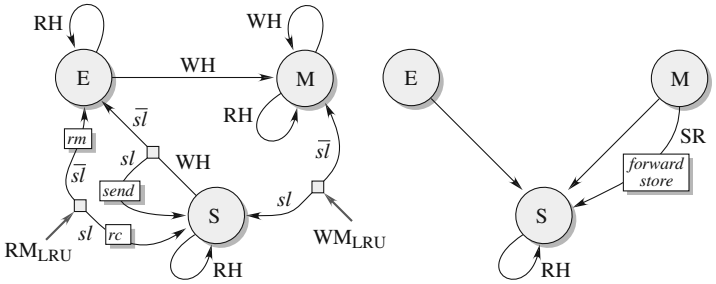


Figure A1.15. *Firefly protocol; broadcasting is ensured by the state of the SharedLine.* rm, rc *and* sl *stand for* read memory, read cache *and* SharedLine*, respectively*

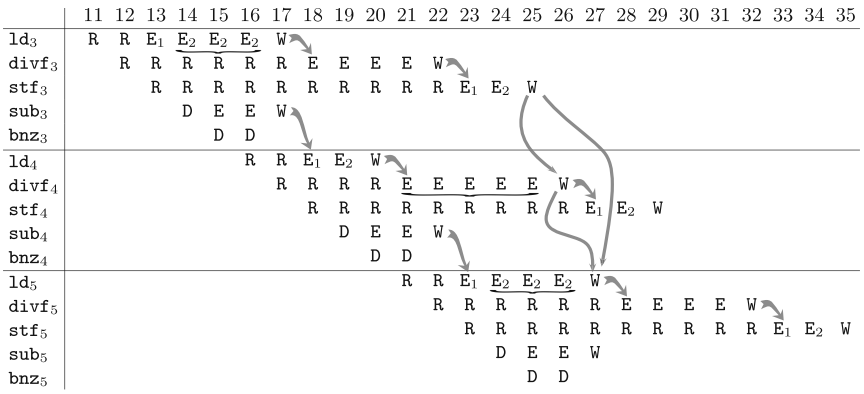
memory (*write-through*, *w-t*). The other caches that have the line are set to *SharedLine* and update the line (*read*).

– A WM is handled like an RM followed by a WH. If the data are present in P_i, it is updated there and the state switches to $\mathbb{S}$. Otherwise, the line in P_0 changes to $\mathbb{M}$.

A line in the cache in the $\mathbb{E}$ or $\mathbb{S}$ state can be replaced at any time, since coherence is ensured.

A1.7. Superscalar architectures

Exercise 13.1. (Periodicity of operation) (page 321) The period of operation is equal to 2. Iterations 2 and 4, on the one hand, and 3 and 5, on the other, lead to identical sequences of steps.



Exercise 13.2. (Examples of hazards) (page 321)

1) Types of conflicts: $I_1 \prec I_2$, $I_2 \prec I_3$, $I_1 \prec I_3$, $I_3 \prec I_4$.

```
I1: divf f2,f2,f1
I2: stf  f2,r0(r3) ; RAW with I1
I3: mulf f2,f5,f6  ; WAR with I2, WAW with I1
I4: stf  f2,r0(r4) ; RAW with I3
```

2) Comment. – Since `f2` is renamed as `f2'` and `f2"`, where `f2'` and `f2"` are the avatars of `f2`, we can write:

```
divf f2',f2,f1 | F  I  E  E  E  E  E  E  W
stf f2',r0(r3) |    F  I  .  .  .  .  .  I  E  W
mulf f2",f5,f6 |       F  I  E  E  W
stf f2",r0(r4) |          F  I  .  I  E  .  W
```

Phase E of the `mulf` can be initiated before the end of the `stf`, since `mulf` can use `f5` and `f6`. The WAR with `stf` is resolved (avatar) and there is no hold in W. The resolution of the WAW between I_1 and I_3 satisfies the WAR.

3) Execution diagram:

```
divf f2',f2,f1 | F  I  E  E  E  E  E  E  W
stf f2',r0(r3) |    F  I  .  .  .  .  .  I  E  W
mulf f2",f5,f6 |       F  I  E  E  W
stf f2",r0(r3) |          F  I  .  I  E  .  .  .  W
```

The only difference with the previous sequence stems from the writing at the same address, which can only be done *in the order* of the program!

Exercise 13.3. (Execution of a loop) (page 322)

Comments of Figure A1.16:

– Arrows 2, 4 and 7: RAW conflict, the `divf` waits for the `ld` to perform its WB.

– Arrow 1: there is only one stage and one pipeline for the `ld` and the `st`. $\texttt{ld}_2$ waits for $\texttt{ld}_1$ to free up the execution stage.

– Arrows 3 and 5: resource problem. There are only two entries d_1 and d_2 in the reservation station. We have to wait for one of the two to become free again.

– Arrow 6: the execution stage of $\texttt{st}_2$ is busy. $\texttt{ld}_3$ is delayed.

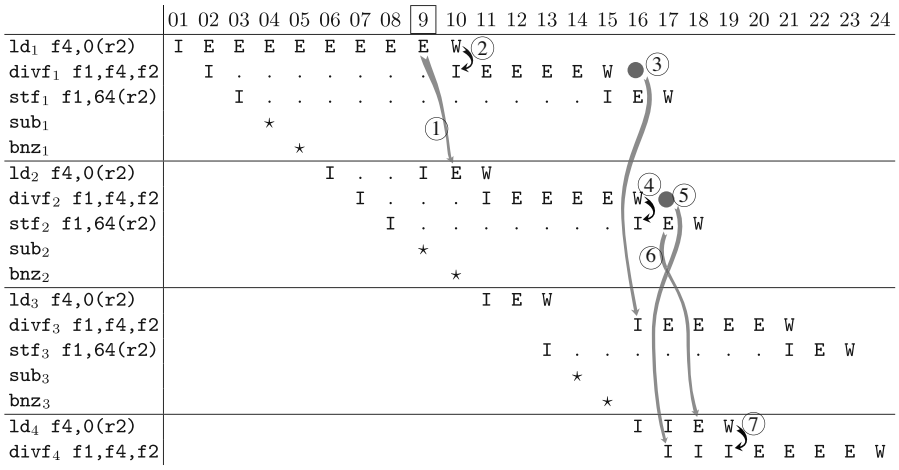


Figure A1.16. *Execution of a loop*