Chapter 17
Answers to Selected Excercises

Problem 1.1.

- `TMS320DM646` is a processor chip from Texas Instruments specialized in Multimedia (video and audio) processing. The chip contains two processors: an ARM RISC core and a Digital Signal Processor (DSP). Writing software for this chip thus means writing a program for each processor; and compiling each program with a compiler for that specific processor. The chip also includes many different peripherals and an extensive on-chip bus system. These peripherals require additional programming.
- `EP4CE115` is a Field Programmable Gate Array from Altera. This chip is programmed with a Hardware Description Language (HDL) such as Verilog or VHDL. These programs are then compiled to the FPGA using a hardware synthesis tool specific for this FPGA.
- `SAM7S512` is a microcontroller from Atmel containing an ARM RISC core, an on-chip bus system, and many peripherals (memory, input/output ports, timers, and so on). Writing software for this chip means writing a program for the ARM processor, and compiling the program with a compiler for that processor.
- `ADSP-BF592` is a DSP processor from Analog Devices which includes multiple peripherals besides the DSP. Such a chip is useful for control applications that require signal processing. Writing software for this chip means writing software for the DSP processor on the chip, and using a compiler specific to that processor.
- `XC5VFX100T` is a Field Programmable Gate Array from Xilinx. This chip is programmed in a HDL, and these programs are then compiler (synthesized) for the FPGA using a hardware synthesis tool specific for this FPGA.

Problem 1.3.

(a) The latency, the time it takes to compute a single output from a single input, will take the same amount of time on the parallel processor. The throughput, the number of inputs that can start processing per time unit, will increase. The throughput is only limited by the slowest processor in the processing pipeline, which will compute the first (fA) or the last (fC) step of the overall function. fA and fC take 40% of the execution time of the original function. Hence, the throughput of the parallel system will increase with a factor $1/0.4 = 2.5$.

(b) The power dissipation of the single processor system is $P = C.V^2.f$, with C a constant. The parallel processor system can scale the voltage and frequency of each processor to 40% of the original value. The power dissipation of the parallel processor system is therefore $P = 3.C.(0.4V)^2.(0.4f)$. The ratio of the single-processor power dissipation to the parallel-processor power dissipation is thus $R = 1/(3.0.4^3) = 5.2$.

Problem 1.5.

This question asks for more than a simple yes/no answer; it asks for the motivation for answering yes or no.

This question can be answered by evaluating the amount of computations that are performed by the CORDIC program. Clearly, for a fixed amount of resources (processors, gates), a larger amount of computations will require a larger amount of computation time. The CORDIC algorithm in Listing 1.6 has 20 iterations. In each iteration, the algorithm performs the following operations.

- a 32-bit addition
- a 32-bit subtraction
- two shift-left over a variable amount of positions
- a 32-bit addition or subtraction
- a lookup table access

These are the main operations; we ignore other activities such as loop counter management. Futhermore, observe in the CORDIC algorithm that the output of one loop iteration (variables X, Y, and current) is used for the next iteration. Hence, the loop iterations depend on one another. We have to compute all of time one after the other.

The question asks if its possible to complete these computations in a fixed amount of clock cycles or time.

- Is it possible to complete the algorithm in 1000 clock cycles? Yes. 1000 clock cycles gives you 50 clock cycles per iteration. A simple processor is able to handle the computational load within that time.
- Is it possible to complete the algorithm in 1000 microseconds? Certainly. Running the above mentioned processor at a clock period of 1 microsecond (only 1 MHz) gives you the require performance.

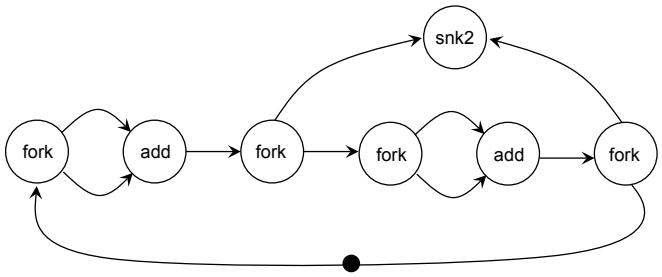
- Is it possible to complete the algorithm in 1 clock cycle? This gets tricky. On a simple processor, definitely not. With a large amount of hardware (lots of adders and subtractors), yes.
- Is it possible to complete the algorithm in 1 microsecond? This would be possible if we could run the hardware mentioned earlier at a clock frequency of 1MHz. However, this is unlikely, because the loop dependencies require us to calculate the adders and shifters one after the other. Thus, we have a chain of at least 20 adder-shifter combinations, and each adder-shifter has only 50ns to compute the output. Therefore, the answer for this question is 'not in current CMOS technology'.

Problem 2.1.

If we observe the stream of tokens accepted by the `snk` actor, we see the values 2, 4, 8, 16, and so on. The value of token n therefore is equal to 2^n.

Problem 2.3.

A loop of two copies of the original dataflow system will produce two tokens that can be fed into the `snk2` actor. The initial token from the original graph should not be duplicated; this would modify the sequence of values observed in `snk2`.



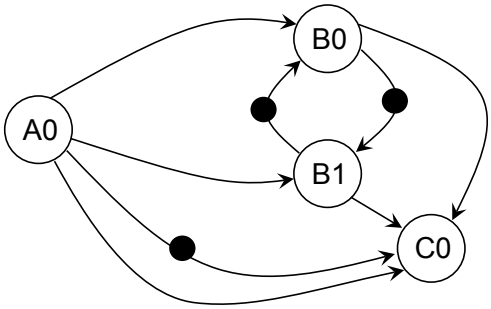
Problem 2.5.

Constructing the topology matrix G, with columns A, B, C, and rows (A,B), (B,C), and (C,A), we find:

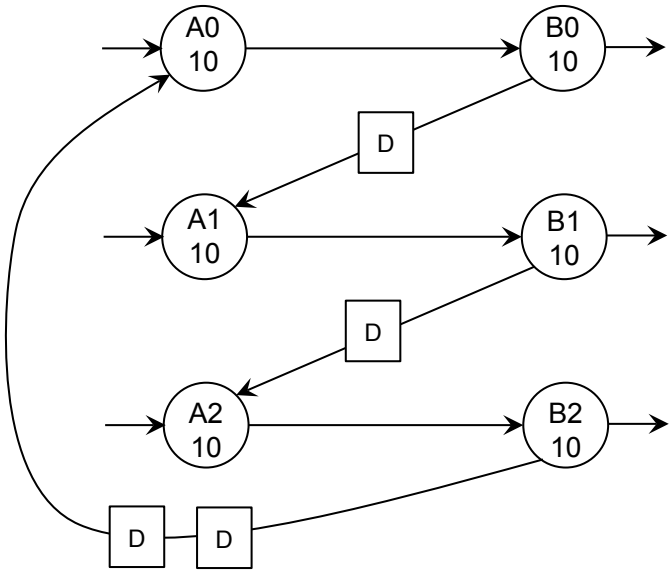
$$G = \begin{bmatrix} X & -2 & 0 \\ 0 & 1 & -Y \\ -1 & 0 & 1 \end{bmatrix}$$

For a PASS to exist, the rank of this matrix needs to be 2, in other words, a combination of two rows can yield the third one. Choosing $X = 2$ and $Y = 1$ gives a possible solution, and in general $X = 2.k$ and $Y = k$ for a positive integer k.

Problem 2.7.



Problem 2.9.



The iteration bound before unfolding is $(10+10)/4 = 5$. The iteration bound after unfolding is $(10+10+10+10+10+10)/4 = 15$.

Problem 3.1.

```
#include <malloc.h>
```

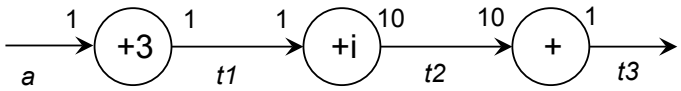
```
typedef struct fifo {
  int size;          // current queue size
  int *data;         // token storage
  unsigned wptr;     // write pointer
  unsigned rptr;     // read pointer
} fifo_t;

void init_fifo(fifo_t *F) {
  F->wptr = F->rptr = 0;
  data = (int *) malloc( 2 * sizeof(int) );
  size = 2;
}

void put_fifo(fifo_t *F, int d) {
  if (((F->wptr + 1) % size) != F->rptr) {
    F->data[F->wptr] = d;
    F->wptr = (F->wptr + 1) % size;
  } else {
    // fifo full - resize
    int newsize = 2*size;
    int *newdata = (int *) malloc( newsize * sizeof(int) );
    unsigned i;
    for (i=0; i<size; i++)
      newdata[i] = data[i];
    for (i=size; i<newsize; i++)
      newdata[i] = 0;
    data = newdata;
    size = newsize;
    put_fifo(F, d); // this call will succeed
  }
}

int get_fifo(fifo_t *F) {
  int r;
  if (F->rptr != F->wptr) {
    r = F->data[F->rptr];
    F->rptr = (F->rptr + 1) % size;
    return r;
  }
  return -1;
}

unsigned fifo_size(fifo_t *F) {
  if (F->wptr >= F->rptr)
    return F->wptr - F->rptr;
  else
    return size - (F->rptr - F->wptr) + 1;
}
```

Problem 3.3.

Let's first write a small C program and construct an equivalent SDF graph. Here is a C program with a single input and a single output and a loop.

```
unsigned myfunc(unsigned a) {
  unsigned i, j;
  unsigned t1, t2, t3 = 0;
  t1 = a + 3;
  for (i=0; i<10; i++) {
     t2 = t1 + i;
     t3 = t3 + t2;
  }
  return t3;
}
```

Assuming we map each addition in the C program to a separate actor, the SDF version of the program would look as follows,



The first actor adds three to the input token and produces a single output token; the second actor adds the numbers 0 to 9 to the input token and produces 10 output tokens. The third actor takes the sum of 10 input tokens and produces a single output token. The loop bound (10) thus translates into actor production/consumption rates. Indeed, the inner loop body in the C program executes 10 times for every execution of the code outside of the loop. Hence, there are 10 values for `t2` produced for every value of `t1`. The multi-rate dataflow diagram thus is a natural rendering of a C program with loops. The loop bounds appear as production/consumption rates, and as a consequence, data-dependent loop bounds will result in variable or unknown production/consumption rates.

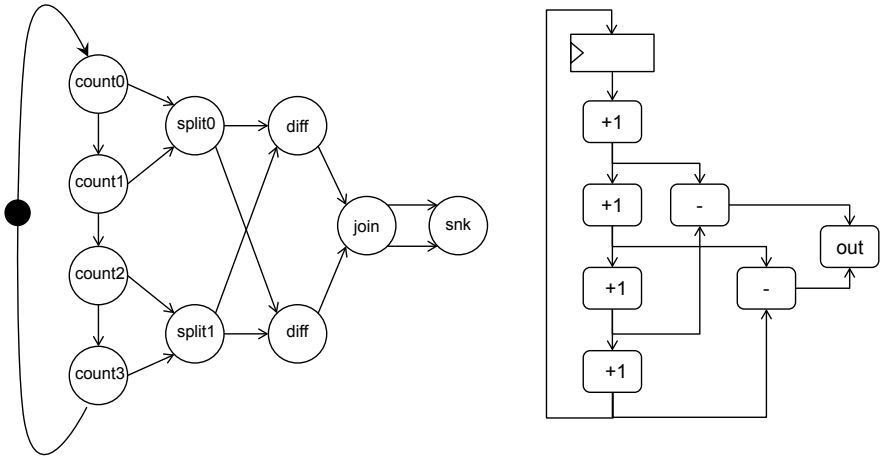
Problem 3.5.

```
void count(actorio_t *g) {
  while (1) {
    if (fifo_size(g->in[0]) >=1 ) {
      unsigned i;
      i = get_fifo(g->in[0]) + 1;
      put_fifo(g->out[0], i);
      put_fifo(g->out[1], i);
    }
    stp_yield();
  }
}

void split(actorio_t *g) {
  while (1) {
    while (fifo_size(g->in[0]) >=2 ) {
```

```
      i = get_fifo(g->in[0]) + 1;
      put_fifo(g->out[0], get_fifo(g->in[0]));
      put_fifo(g->out[1], get_fifo(g->in[0]));
    }
    stp_yield();
  }
}

void diff(actorio_t *g) {
  while (1) {
    while (fifo_size(g->in[0]) >=2 ) {
      put_fifo(g->out[0], get_fifo(g->in[0]) -
                          get_fifo(g->in[0]));
    }
    stp_yield();
  }
}

void join(actorio_t *g) {
  while (1) {
    while ( (fifo_size(g->in[0]) >=1) &&
            (fifo_size(g->in[1]) >=1) ) {
      put_fifo(g->out[0], get_fifo(g->in[0]));
      put_fifo(g->out[0], get_fifo(g->in[1]));
    }
    stp_yield();
  }
}

void snk(actorio_t *g) {
  while (1) {
    while ( fifo_size(g->in[0]) >=1 ) {
      printf("%d\n", get_fifo(g->in[0]));
    }
    stp_yield();
  }
}

int main() {
  fifo_t q1, q2, q3, q4, q5, q6, q7;
  fifo_init(&q1);
  fifo_init(&q2);
  fifo_init(&q3);
  fifo_init(&q4);
  fifo_init(&q5);
  fifo_init(&q6);
  fifo_init(&q7);

  actorio_t count_io   = {{&q1}, {&q1,&q2}};
  actorio_t split_io   = {{&q2}, {&q3,&q4}};
  actorio_t diff1_io   = {{&q3}, {&q5}};
  actorio_t diff2_io   = {{&q4}, {&q6}};
  actorio_t join_io    = {{&q5,&q6}, {&q7}};
  actorio_t snk_io     = {{&q7}};
```

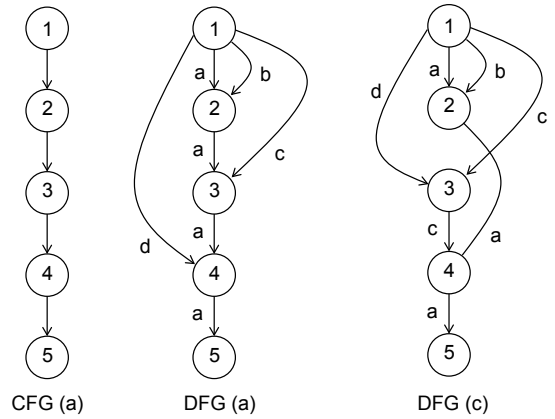
```
  stp_create(count, &count_io); // create thread
  stp_create(split, &split_io); // create thread
  stp_create(diff,  &diff1_io); // create thread
  stp_create(diff,  &diff2_io); // create thread
  stp_create(join,  &join_io ); // create thread
  stp_create(snk,   &snk_io  ); // create thread

  stp_start(); // run the schedule

  return 0;
}
```

Problem 3.7.



This circuit does not have a particular purpose or function; its only use is to demonstrate the conversion of a multi-rate data-flow graph into hardware.

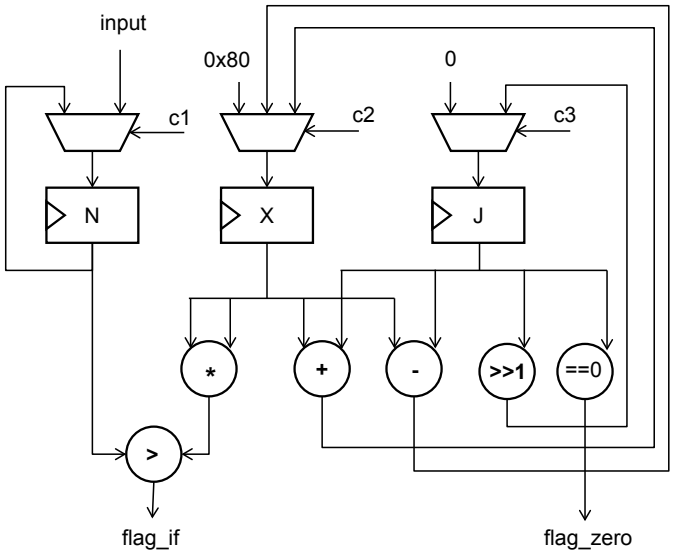
Problem 4.1.



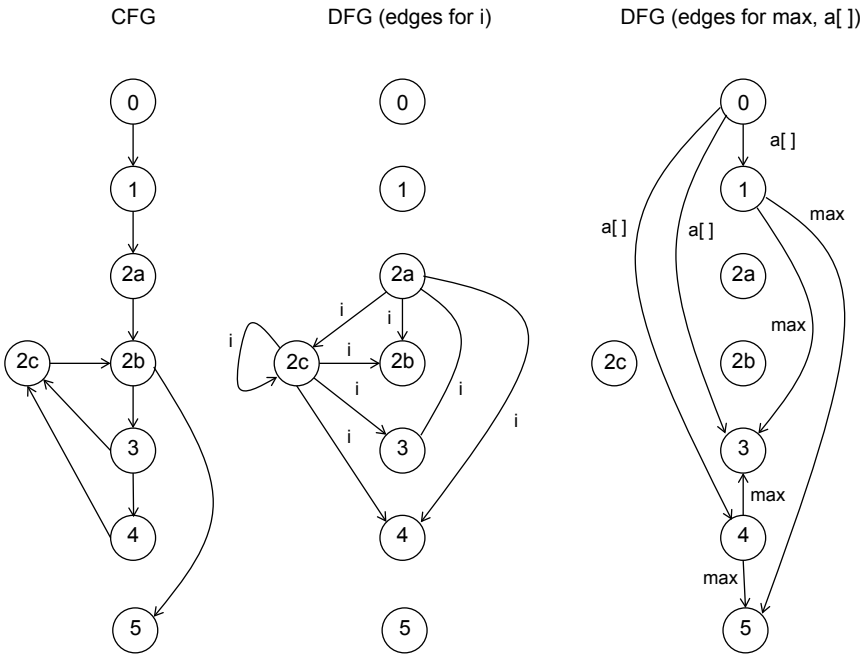
(a) Refer to the Figure.
(b) The longest path is 4.
(c) The function can be rewritten as shown below. The resulting DFG is shown in the Figure. The longest path in the optimized DFG is 3.

```
int addall(int a, int b, int c, int d) {
  a = a + b;
  c = c + d;
  a = a + c;
  return a;
}
```

Problem 4.2.



Problem 4.5.



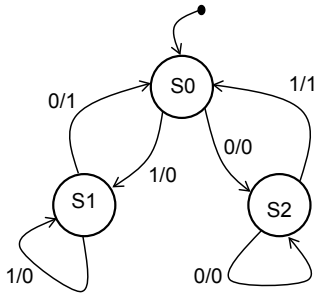
Problem 4.7.

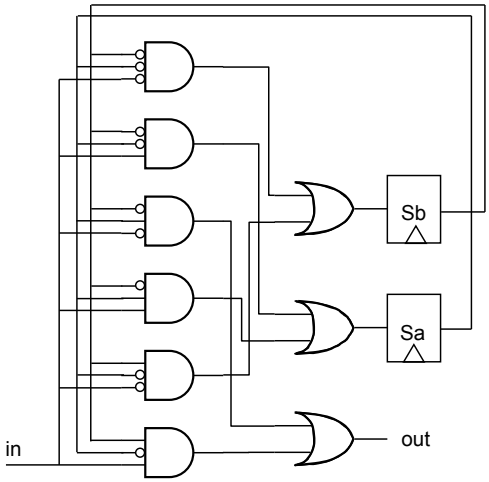
```
unsigned char mysqrt(unsigned int N) {
  unsigned int x1, x2, x3, j1, j2;
  x1 = 0;
  for (j1 = 1<<7; merge(j1,j2) != 0; j2 = merge(j1, j2) >> 1) {
    x2 = merge(x1, x2, x3) + merge(j1, j2);
    if (x2 * x2 > N)
       x3 = x2 - merge(j1, j2);
  }
  return merge(x2, x3);
}
```

Problem 5.1.

Circuits c and d.

Problem 5.3.





Problem 5.5.

```
dp divider(in  x     : ns(8);
           in  y     : ns(8);
           in  start : ns(1);
           out q     : ns(10);
           out r     : ns(8);
           out done  : ns(1)) {
  reg rq  : ns(10);
  reg rr  : ns(8);
  reg yr  : ns(8);
  reg str : ns(1);
  reg cnt : ns(4);
  sig z   : ns(10);
  sig qb  : ns(1);
  sig a,b : ns(8);
  sig rb  : ns(8);

  always      { q   = rq;
                r   = rr;
                str = start;
              }
  sfg iterate { z  = 2*a - b;
                qb = (z > 0x7f) ? 0 : 1;
                rb = (z > 0x7f) ? 2*a : z;
              }
  sfg init    { rq  = 0;
                rr  = x;
                yr  = y;
                cnt = 8;
              }
  sfg go      { a   = rr;
                b   = yr;
                rq  = (rq << 1) | qb;
                rr  = rb;
```

```
                cnt = cnt - 1;
              }
  sfg busy    { done = 0;
              }
  sfg hasdone { done = 1;
              }
}
fsm c_divider(divider) {
  initial s0;
  state s1, s2, s3;
  @s0 if (str) then (busy, iterate, go) -> s1;
               else (busy, init) -> s0;
  @s1 if (cnt == 0) then (hasdone) -> s0;
                    else (busy, iterate, go) -> s1;
}

dp dividertest {
  sig x, y : ns(8);
  sig start : ns(1);
  sig q : ns(10);
  sig r : ns(8);
  sig done : ns(1);

  use divider(x, y, start, q, r, done);
  always   { $display($cycle, " ", x, " ", y, " ", start,
                              " ", q, " ", r, " ", done);
           }
  sfg go   { x     = 12;
             y     = 15;
             start = 1;
           }
  sfg wait { x = 0;
             y = 0;
             start = 0;
           }
}
fsm c_dividertest(dividertest) {
  initial s0;
  state s1;
  @s0 (go) -> s1;
  @s1 (wait) -> s1;
}

system S {
  dividertest;
}
```

The simulation output illustrates that the result is obtained in 10 cycles:

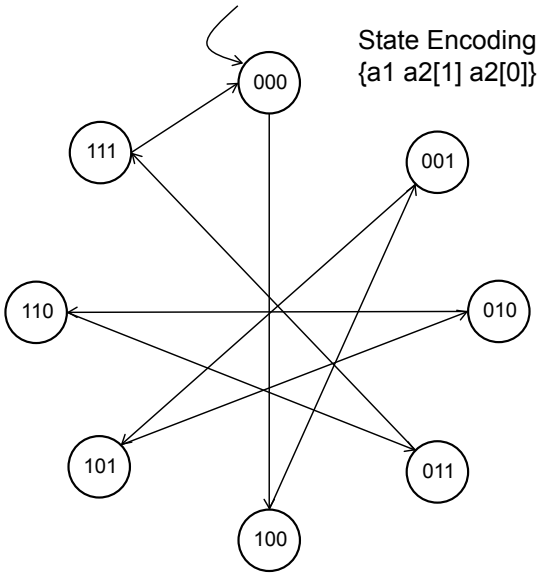
```
> /opt/gezel/bin/fdlsim div.fdl 15
0 c f 1 0 0 0
1 0 0 0 0 c 0
2 0 0 0 1 9 0
3 0 0 0 3 3 0
```

```
  4  0  0  0  6  6  0
  5  0  0  0  c  c  0
  6  0  0  0 19  9  0
  7  0  0  0 33  3  0
  8  0  0  0 66  6  0
  9  0  0  0 cc  c  1
```

Problem 5.8.

```
dp filter(in a : ns(32); out q : ns(32)) {
  reg t0, t1, t2, t3, t4;
  always {
     t4 = a;
     t3 = t4;
     t2 = t3;
     t1 = t2;
     t0 = t1;
     q = -t0 + 5*t1 + 10*t2 + 5*t3 - t4;
  }
}
```

Problem 5.10.



Problem 6.1.

Instruction	M1	M2	ALU	Horizontal	Vertical
	M1[1:0]	M2[1:0]	ALU[1:0]	H[5:0]	V[2:0]
SWAP	10	10	00	101000	001
ADD R1	01	00	00	010000	010
ADD R2	00	01	00	000100	011
COPY R1	00	10	00	001000	100
COPY R2	10	00	00	100000	101
NOP	00	00	00	000000	000

- The encoding of the horizontal micro-instruction corresponds to `M1 M2 ALU`.
- The encoding of the vertical micro-instruction implies an additional decoder that transforms the vertically encoded micro-operation into a horizontally encoded micro-operation. For example, the msbit of `M1` can be obtained as `M1[1] = (Ṽ[2] & Ṽ[1] & V[0]) | (V[2] & Ṽ[1] & V[0])`.

Problem 6.3.

Combination b0 b1 b2 b3	Meaning
Instruction 1 1 X 0 0	JUMP RELATIVE (`CSAR = CSAR+offset;`)
Instruction 2 X 1 1 0	JUMP ABSOLUTE (`CSAR = offset;`)
Instruction 3 0 1 1 1	CALL SUBROUTINE (`RETURN = CSAR+1; CSAR = offset;`)
Instruction 4 X 0 1 0	RETURN FROM SUBROUTINE (`CSAR = RETURN;`)

Problem 6.5.

(a) No

(a) Yes, assuming careful initialization of CSAR, and assuming the datapath state can be safely shared between the two different threads.

Problem 7.1.

```
int main() {
  int a, b;

  if (((unsigned) &a) < ((unsigned) &b))
    printf("Stack␣grows␣upwards\n");
  else
    printf("Stack␣grows␣downwards\n");
}
```

Problem 7.4.

a Not correct

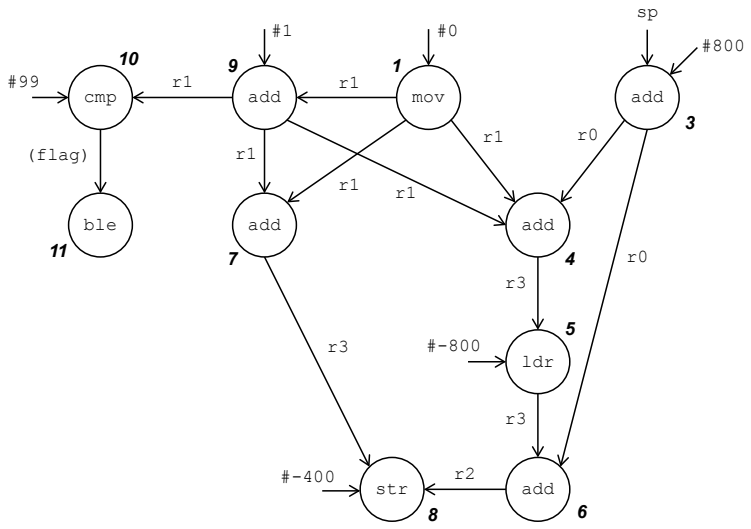
b Memory-mapped coprocessor registers are registers that are visible as memory locations on the processor. Accessing such a register requires memory-load and/or memory-store operations. It is therefore very different from accessing a processor register. The storage specifier `register` only applies to processor registers.

Problem 7.5.

a The compiler computed the result of the for loop at compile time. Such 'unrolling' is possible when compile-time optimization is used. In this case, the optimization flag `-O2` was used. Close inspection of the assembly listing shows that the instructions compute the value `5*a + 6`. The same value can be obtained by symbolically computing the result of the C function in Listing 7.13.

b Using a similar argument as for part (a) of the question, we can derive that the result of the computation would be `5*a - 6`. Hence, the assembly code would remain the same apart from the last instructions: instead of an `add` instruction, we would expect a `subtract` instruction.

Problem 7.7.



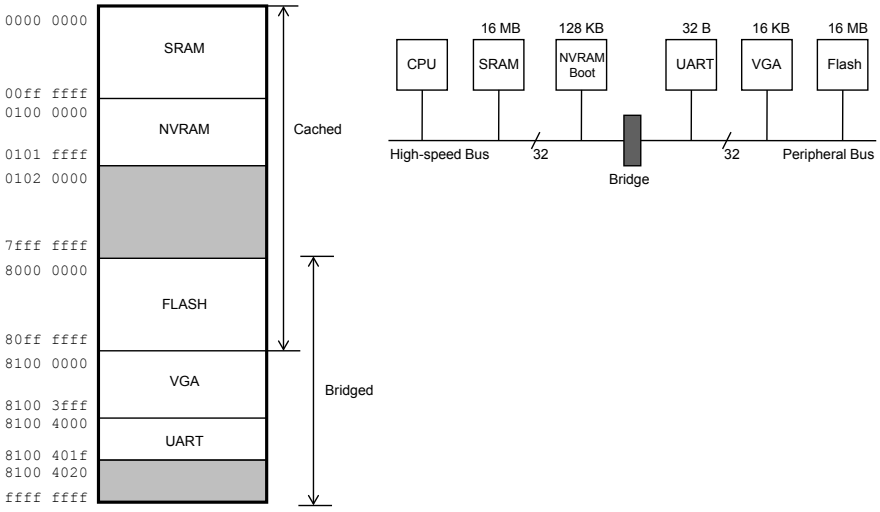
a

b Instructions 1, 3, 4, and 9 can all be seen to have a dependency into the memory-load instruction, and are thus involved in the address calculation.

Problem 8.1.

a Every access to the UART needs to be done directly to the peripheral. Caching those variables makes no sense. This is particularly true for variables (memory locations) that can be updated outside of the control of the processor. An example of such a register is the data-receive register of the UART (See Section 8.4.3).
b Multiple processors can update memory locations independently from one another. Whenever a given memory location is updated (written), all cached copies of that memory location should be invalidated. This problem is known as the cache-coherency problem: multiple caches on a bus must maintain a consistent view of the main memory.
c The data cache, since the instruction memory-space is read-only.

Problem 8.3.



Problem 9.1.

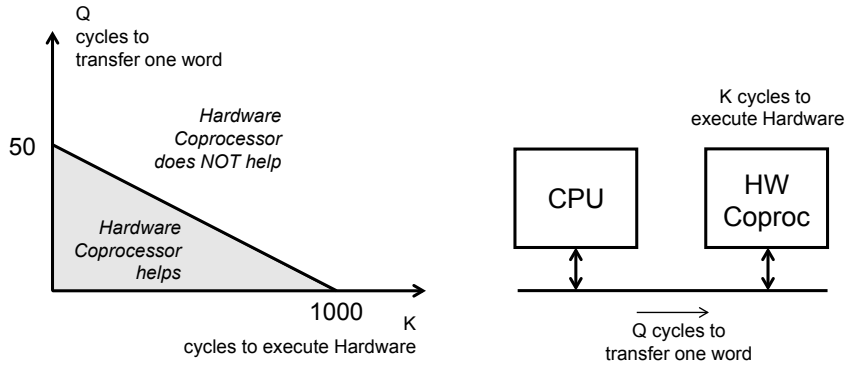
To communicate from CPU1 to CPU2, at least the following bus transactions will be needed:

- A high-speed bus transaction (50ns)
- A bridge transfer (100 ns)
- A low-speed bus transaction (200ns)

The communication latency is therefore $50 + 100 + 200ns$, or $350ns$, or 2.87M transactions per second. This is an optimistic upperbound; in practice additional memory accesses will slow down the system operation.

Problem 9.3.

By plotting the equation $1000 = K + 20Q$, the (K, Q) plane is divided in two regions. In one region, the sum $K + 20Q$ is smaller than 1000. This means that the total execution time consumed by hardware is smaller than the execution time for the same functionality in software. Hence, the hardware coprocessor provides an overall gain in speedup. At the other side of the line, the sum $K + 20Q$ is larger than 1000. In this case, the hardware coprocessor does not provide any benefit over the software.



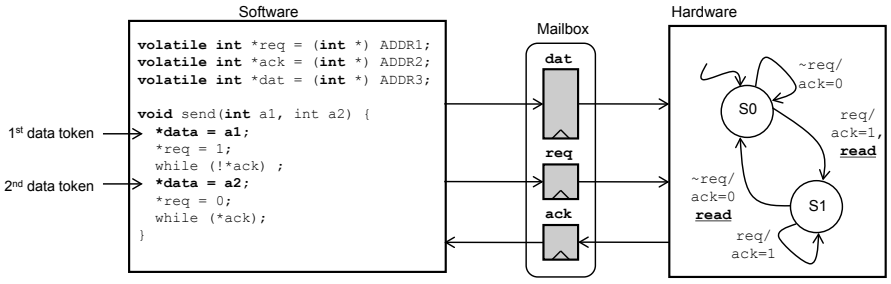
Problem 10.2.

(a) Cursor position 'X' is a memory read.
(b) The address falls within the address range of `a`, which goes from `0x44001084` to `0x44001084 + 0x40*4 = 0x44001184`. Hence, the memory read at cursor position 'X' must be a data memory read.
(c) Address `0x4400111C` corresponds to the address of `a[0x1A]`, or `a[26]`. According to Listing 10.1, `a[26]` is read when `i` must be 27.

Problem 10.4.

(a) Bus master 1 has priority over bus master 2. This can be observed at clock edge 2.
(b) At clock edge 3, bus master 2 gains control over the bus. Hence, bus master 2 controls the bus between clock edge 3 and clock edge 4.
(c) A bus arbiter generates bus-grant signals.
(d) The acknowledge signal is generated by the bus slave to indicate completion of the bus transfer.

Problem 11.2.

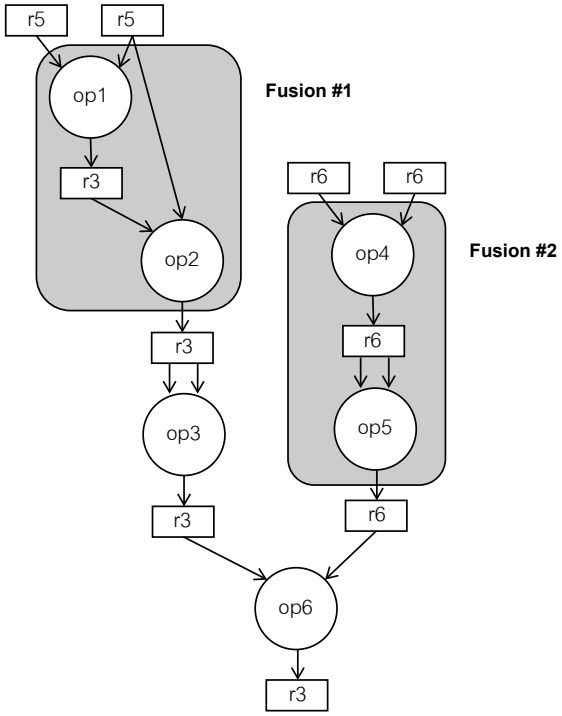


Problem 11.4.

The following solution forces all data to go through the register `loopbackreg`. The module keeps track of the register status through the `rfull` flag. When `rfull` is low (`loopbackreg` is empty), only `write` can set it. When `rfull` is high (`loopbackreg` is filled), only `read` can clear it.

```
dp loopback(in  wdata  : ns(32);
            in  write  : ns(1);
            out full   : ns(1);
            out rdata  : ns(32);
            out exists : ns(1);
            in  read   : ns(1)) {
   reg loopbackreg : ns(32);
   reg rfull       : ns(1);
   always {
     rfull       = (~rfull & write) | (rfull & ~read);
     exists      = rfull;
     full        = rfull;
     rdata       = loopbackreg;
     loopbackreg = (~rfull & write) ? wdata : loopbackreg;
   }
}
```

Problem 11.6.



Problem 11.8.

(a) Yes, the energy consumption per sample is proportional to the amount of work done per sample.
(b) Yes, the custom instruction shortens the execution time, while the question suggests that the added power is the same. Hence the energy per sample decreases.
(c) No. This will increase the power dissipation, and proportionally shorten the execution time. Hence, the energy per sample remains (approximately) the same.
(d) Impossible to decide. The question does not state if the clock frequency remains the same or not. If it remains the same, the energy per sample will decrease because the power dissipation decreases under the same execution time. If the clock frequency is reduced together with the voltage, the benefit from reduced power dissipation is reduced by increased execution time.

Problem 12.3.

(a) The coprocessor has three data input-ports, one data output-port.

(b) The median is computed in a single clock cycle (`dp median`). However, the hardware interface can provide only a single data input and a single data output. Close examination of the hardware interface (`dp medianshell`) shows that this is a time-multiplexed interface. Hence, the design is communication-constrained.

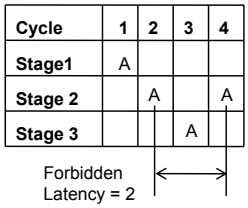
(c) The register structure that holds the three input operands is advanced as follows. First, a zero must be written in the input register of the hardware interface. Next, a non-zero value must be written. This sequence must be repeated three times, to provide each of the three input operands. After the third time, the result can be immediately retrieved from the coprocessor output, because the result is computed in a single clock cycle.

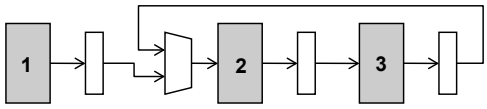
(d) The program is shown below.

```
#include <stdio.h>

int main() {
  unsigned volatile *din  = (unsigned *) 0x80000000;
  unsigned volatile *dout = (unsigned *) 0x80000004;

  *din = 0;
  *din = 36;

  *din = 0;
  *din = 99;

  *din = 0;
  *din = 58;

  printf("The result is %d\n", *dout);
}
```

Problem 12.5.

By analyzing the reservation table, we find that the set of forbidden latencies is (2), because pipeline stage 2 is reused with a latency of 2 cycles. By executing a few pipeline executions, we note that, for optimal utilization of the second pipeline stage, data must enter the pipeline at irregular intervals. The following figure illustrates this.

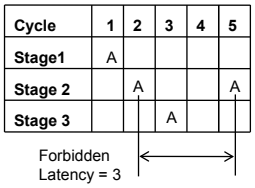


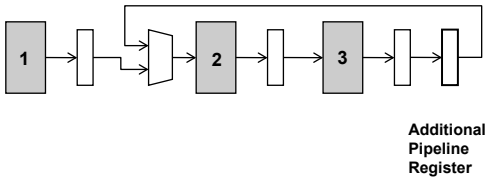


Irregular Pipeline Initiation

Cycle	1	2	3	4	5	6	7	8	9
Stage1	A	B			C	D			
Stage 2		A	B	A	B	C	D	C	D
Stage 3			A	B			C	D	

To address this issue, we need to rework the delays in the pipeline. We cannot change the execution sequence of pipeline stages: it must remains (stage 1, stage 2, stage 3, stage 2). However, we can postpone execution of a pipeline stage by introducing additional pipeline registers. This increases the overall latency of the instruction, but it may obtain a regular pipeline initiation. The following figure illustrates that, by doubling the pipeline register at the output of stage 3, we obtain the desired effect.





Regular Pipeline Initiation

Cycle	1	2	3	4	5	6	7	8	9	10	11
Stage1	A		B		C		D				
Stage 2		A		B	A	C	B	D	C		D
Stage 3			A		B		C		D		