ISA details

No special modifications have been made to the proposed ISA. If anything, the only point of note is that the write port for the register file has indeed been kept as the first operand of all the R-type instructions (as opposed to having ports #1 and #2 be the more conventional `Rs` or `Rt`, with the third port taking the place of `Rd`) and the corresponding M-type instructions in `ADDI` and `LD{B,W}`. Notice that, because of this terminological simplification, R1 and R3 must be muxed for instructions `ST{B,W}` and BEQ (notice the *"s.t. r1 is moved to r3"* comment, see mux `R1R3SRC` within the diagram figure of the processor). The `SLT` instruction is a byproduct of the ALU block's implementation and is rarely used, if at all, in the tested code and the performance tests. The `BEQ` instruction shares the same encoding as M-type instructions, while the `JMP` instruction does its own thing.

Instructions

0x0	R	ADD	r1, r2, r3	// r1 <- r2 + r3
0x1	R	SUB	r1, r2, r3	// r1 <- r2 - r3
0x2	R	AND	r1, r2, r3	// r1 <- r2 AND r3
0x3	R	OR	r1, r2, r3	// r1 <- r2 OR r3
0x4	R	SLT	r1, r2, r3	// r1 <- "1" if r2 < r3, otherwise r1 <- "0"
0x5	R	MUL	r1, r2, r3	// r1 <- r2 * r3
0x7	M	ADDI	r1, r2, imm	// r1 <- r2 + imm
0x10	M	LDB	r1, r2, imm	// r1 <- mem[r2 + imm]
0x11	M	LDW	r1, r2, imm	// r1 <- mem[r2 + imm]
0x12	M	STB	r1, r2, imm	// mem[r2 + imm] <- r1, s.t. r1 is moved to r3
0x13	M	STW	r1, r2, imm	// mem[r2 + imm] <- r1, s.t. r1 is moved to r3
0x30	B	BEQ	r1, r2, imm	
				// if r1 == r2, PC = PC_plus4 + imm, s.t. r1 is moved to r3
0x31	B	JMP	imm	
				// PC = concat(PC(31 downto 27, imm, "00"), imm is instr(24 downto 0)

Encoding details

[R-type]

31-25	24-20	19-15	14-10	9-0
opcode	dst	src1	src2	zeroes

[M-type]

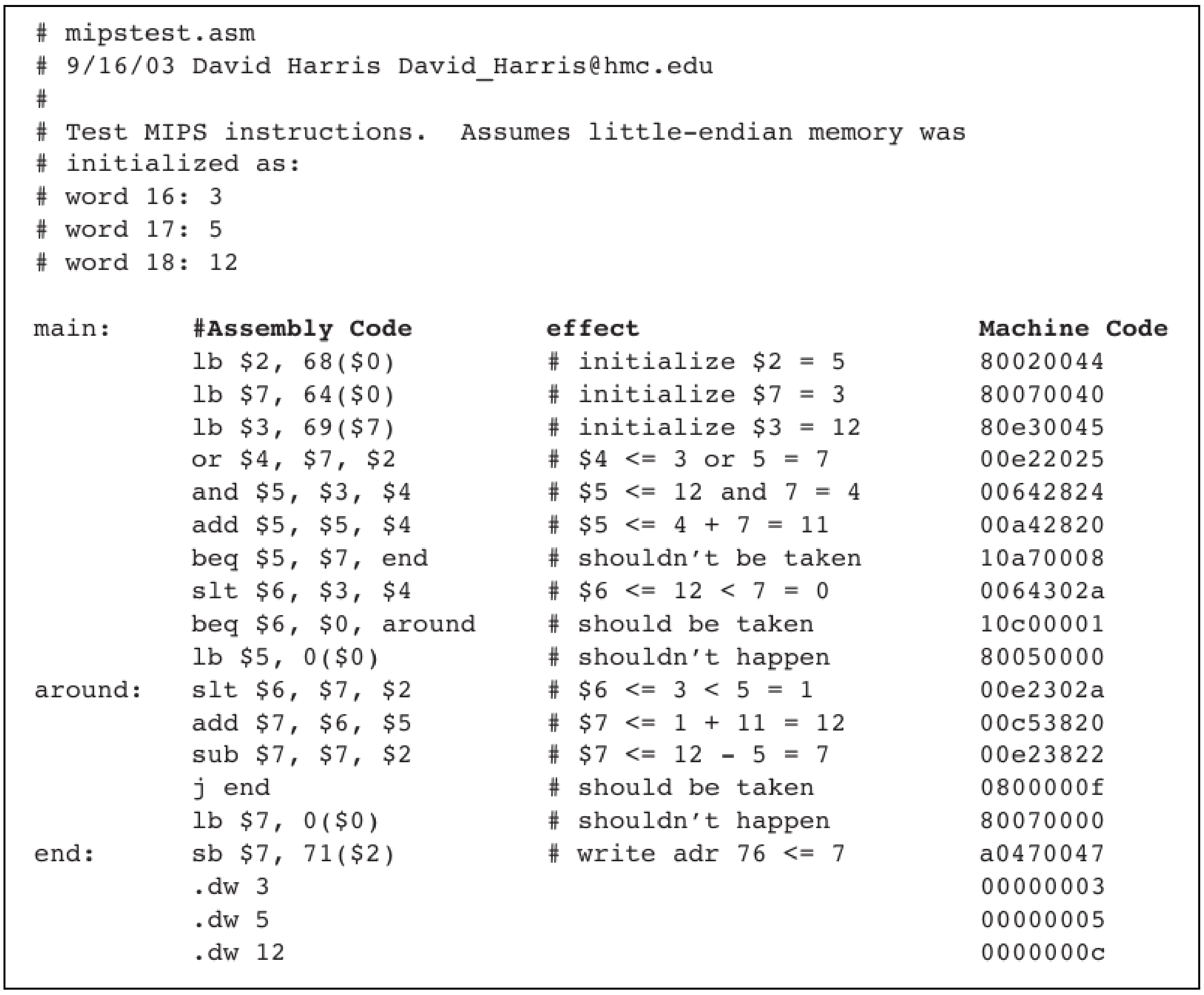
31-25	24-20	19-15	14-0
opcode	dst	src1	offset

[BEQ-type]	opcode_31-25	dst_24-20	src1_19-15	offset_14-0
[JMP-type]	opcode_31-25	offset_24-0		

Sanity-check program and n-th Fibonacci number calculation

The following programs have been used to verify the correct functionality of the processor in its two stages (single cycle and pipelined). Both of them have been pulled from the [CMOS VLSI Design: A Circuits and Systems Perspective - 4th Edition - Weste, Harris] book, where an introductory section of the book (more specifically, chapter 1.7) uses a simple multicycle MIPS to explore the concept *“hierarchical design”*, as well as making a demonstration of what its actual synthetised, physical implementation looks like (refer to chapters 1.9 and 1.10, *“Circuit design”* and *“Physical design”* for further information). Along with a high-level block diagram explaining how the processor is interconnected come the two programs in question: a simple instruction checker and a n-th Fibonacci number calculator. Figures from both the book and the used assembly code are included. The Fibonacci program is also used as a performance test comparing the single cycle and the pipelined versions of the processor. The actual assembly code that is loaded into the memory .dat files resides within the `asmtoisa.py` script.

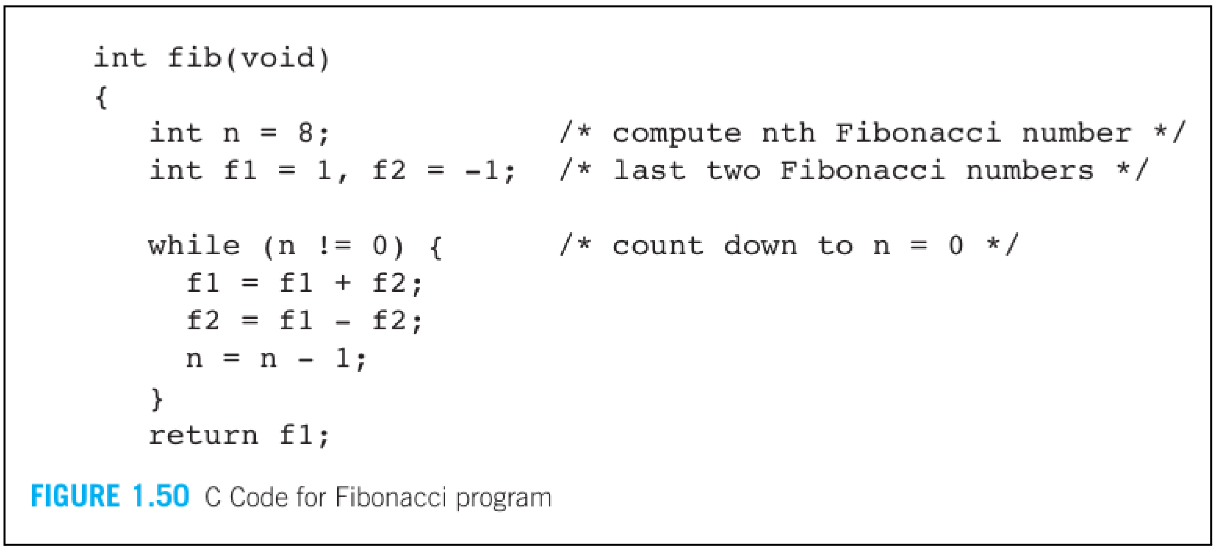
Book figure 1



```
# mipstest.asm
# 9/16/03 David Harris David_Harris@hmc.edu
#
# Test MIPS instructions.  Assumes little-endian memory was
# initialized as:
# word 16: 3
# word 17: 5
# word 18: 12

main:     #Assembly Code        effect                       Machine Code
          lb $2, 68($0)         # initialize $2 = 5          80020044
          lb $7, 64($0)         # initialize $7 = 3          80070040
          lb $3, 69($7)         # initialize $3 = 12         80e30045
          or $4, $7, $2         # $4 <= 3 or 5 = 7           00e22025
          and $5, $3, $4        # $5 <= 12 and 7 = 4         00642824
          add $5, $5, $4        # $5 <= 4 + 7 = 11           00a42820
          beq $5, $7, end       # shouldn't be taken         10a70008
          slt $6, $3, $4        # $6 <= 12 < 7 = 0           0064302a
          beq $6, $0, around    # should be taken            10c00001
          lb $5, 0($0)          # shouldn't happen           80050000
around:   slt $6, $7, $2        # $6 <= 3 < 5 = 1            00e2302a
          add $7, $6, $5        # $7 <= 1 + 11 = 12          00c53820
          sub $7, $7, $2        # $7 <= 12 - 5 = 7           00e23822
          j end                 # should be taken            0800000f
          lb $7, 0($0)          # shouldn't happen           80070000
end:      sb $7, 71($2)         # write adr 76 <= 7          a0470047
          .dw 3                                              00000003
          .dw 5                                              00000005
          .dw 12                                             0000000c
```

Extracted from section A.12.1 of the book, final output of the program should place a 7 in address 76

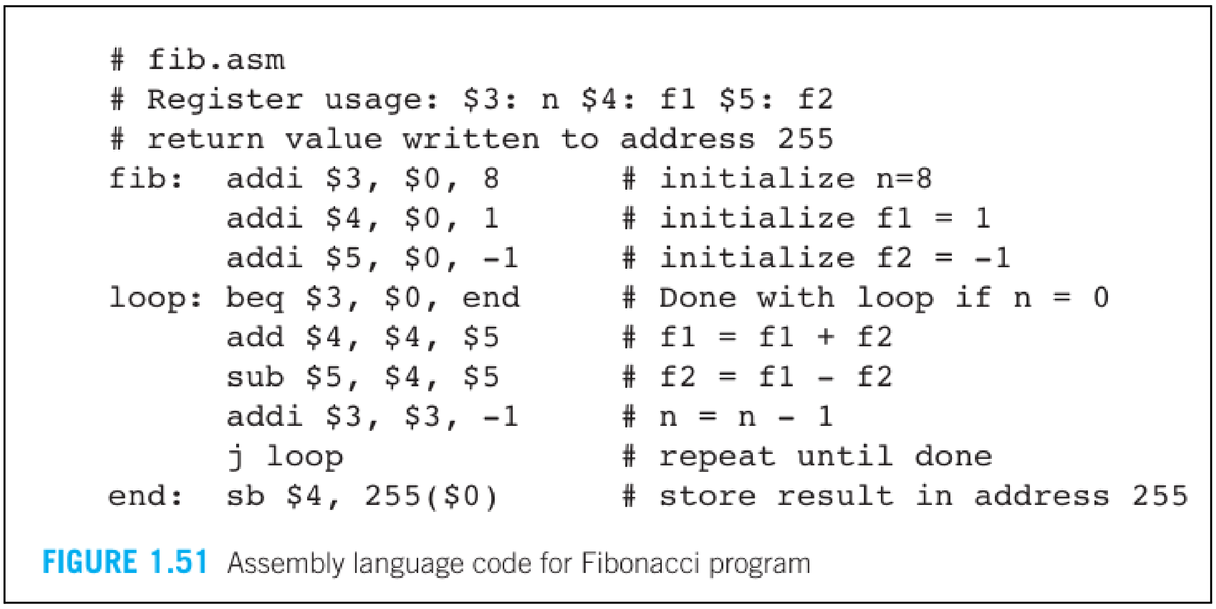
Book figure 2



```
int fib(void)
{
   int n = 8;              /* compute nth Fibonacci number */
   int f1 = 1, f2 = -1;    /* last two Fibonacci numbers */

   while (n != 0) {        /* count down to n = 0 */
     f1 = f1 + f2;
     f2 = f1 - f2;
     n = n - 1;
   }
   return f1;
```

FIGURE 1.50 C Code for Fibonacci program

Book figure 3



```
# fib.asm
# Register usage: $3: n $4: f1 $5: f2
# return value written to address 255
fib:  addi $3, $0, 8       # initialize n=8
      addi $4, $0, 1       # initialize f1 = 1
      addi $5, $0, -1      # initialize f2 = -1
loop: beq $3, $0, end      # Done with loop if n = 0
      add $4, $4, $5       # f1 = f1 + f2
      sub $5, $4, $5       # f2 = f1 - f2
      addi $3, $3, -1      # n = n - 1
      j loop               # repeat until done
end:  sb $4, 255($0)       # store result in address 255
```

FIGURE 1.51 Assembly language code for Fibonacci program

Code figure 1

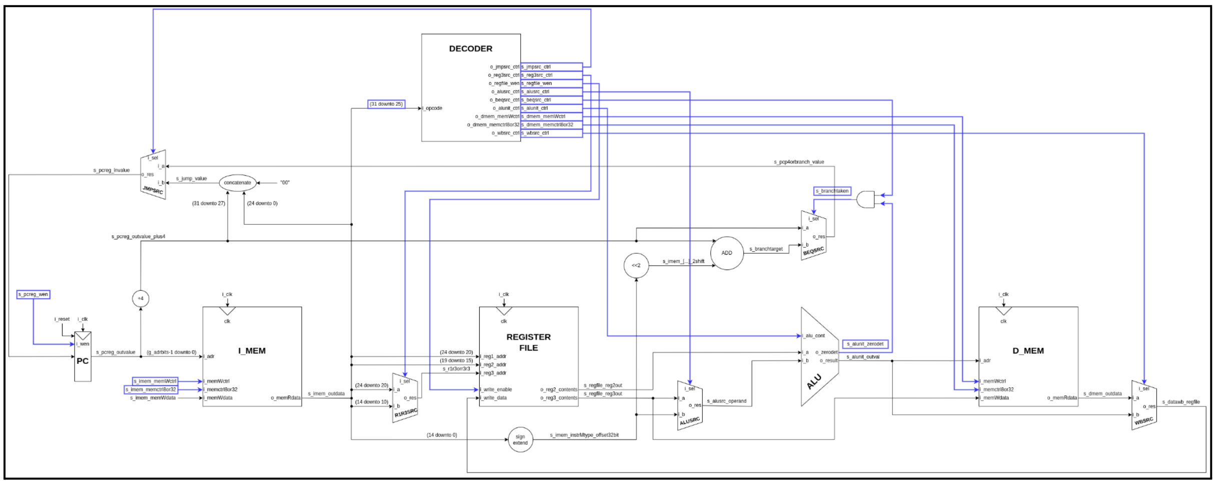
```
main:       #Assembly Code          effect                      Machine Code    Data memory    Bytenum
            LDB 2 0 68              # initialize $2 = 5         20200044        80020044       0
            LDB 7 0 64              # initialize $7 = 3         20700040        80070040       4
            LDB 3 7 69              # initialize $3 = 12        20338045        80e30045       8
            OR 4 7 2                # $4 <= 3 or 5 = 7          06438800        00e22025       12
            AND 5 3 4               # $5 <= 12 and 7 = 4        04519000        00642824       16
            ADD 5 5 4               # $5 <= 4 + 7 = 11          00529000        00a42820       20
            BEQ 5 7 8   (end)       # shouldn't be taken        60538008        10a70008       24
            SLT 6 3 4               # $6 <= 12 < 7 = 0          08619000        0064302a       28
            BEQ 6 0 1   (around)    # should be taken           60600001        10c00001       32
            LDB 5 0 0               # shouldn't happen          20500000        80050000       36
around:     SLT 6 7 2               # $6 <= 3 < 5 = 1           08638800        00e2302a       40
            ADD 7 6 5               # $7 <= 1 + 11 = 12         00731400        00c53820       44
            SUB 7 7 2               # $7 <= 12 - 5 = 7          02738800        00e23822       48
            JMP 15                  # should be taken           6200000f        0800000f       52
            LDB 7 0 0               # shouldn't happen          20700000        80070000       56
end:        STB 7 2 71              # write adr 76 <= $7        24710047        a0470047       60
            STW 7 0 0               # mem(0) <= $7 word-wise    26700000        00000003       64
            STB 7 0 1               # mem(1) <= $7 byte-wise    24700001        00000005       68
            LDB 1 0 1               # $1 <= mem(1) byte-wise    20100001        0000000c       72
            LDW 1 0 0               # $1 <= mem(0) word-wise    22100000                       76
            ADDI 3 3 -1             # $3 <= $3 + -1             0e31ffff                       80
            MUL 3 3 3               # $3 <= $3 * $3             0a318c00                       84
```

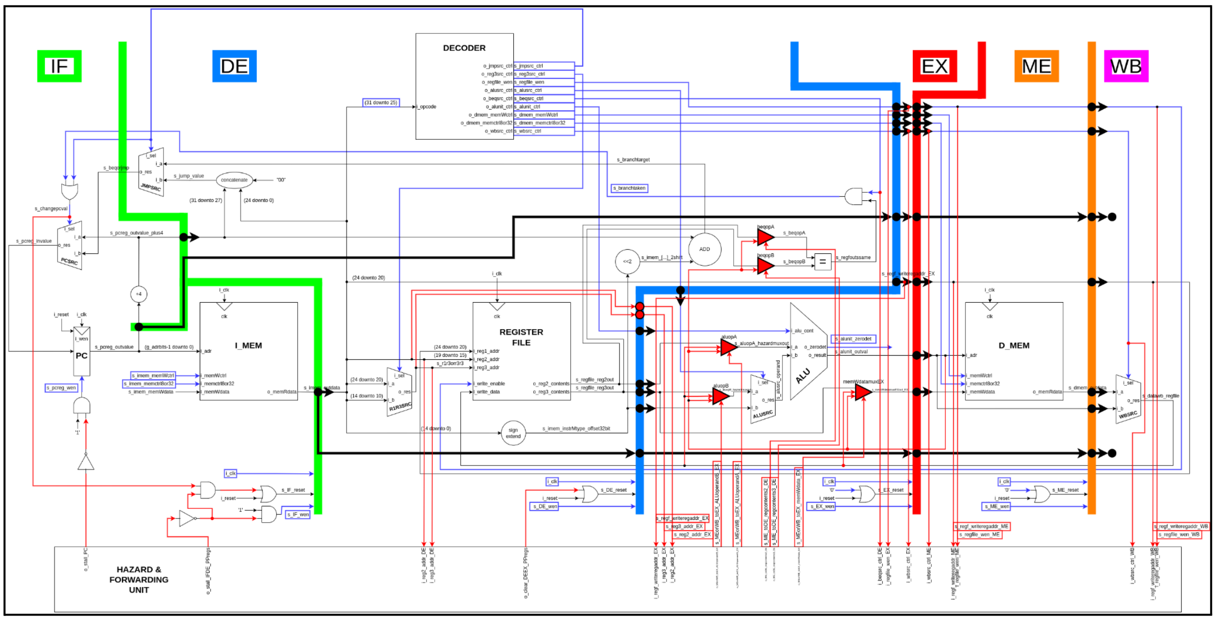
Code figure 2

```
# fib.asm
# Register usage: $3: n $4: f1 $5: f2
# return value written to address 255
fib:    addi $3, $0, 8      # initialize n=8
        addi $4, $0, 1      # initialize f1 = 1
        addi $5, $0, -1     # initialize f2 = -1
loop:   beq $3, $0, end     # Done with loop if n = 0
        add $4, $4, $5      # f1 = f1 + f2
        sub $5, $4, $5      # f2 = f1 - f2
        addi $3, $3, -1     # n = n - 1
        j loop              # repeat until done
end:    sb $4, 255($0)      # store result in address 255

    ##############################
    # translation to "bespoke" format
    ##############################
    fib:    ADDI 3 0 8      # initialize n=8
            ADDI 4 0 1      # initialize f1 = 1
            ADDI 5 0 -1     # initialize f2 = -1
    loop:   BEQ 3 0 4       # Done with loop if n = 0
            ADD 4 4 5       # f1 = f1 + f2
            SUB 5 4 5       # f2 = f1 - f2
            ADDI 3 3 -1     # n = n - 1
            JMP 3           # repeat until done
    end:    STB 4 0 255     # store result in address 255
```

Processor diagrams

All the signals names correspond to the actual implementation in code. See the .drawio file for further reference. The single cycle implementation has been used as a baseline to confirm the results of the pipelined version and a comparison of both regarding the performance tests is provided. The relatively low level of idiosyncrasies and the (much to my chagrin) simplicity of the pipeline design lends itself to being more easily explained via visual queues, or at least that would be my take on the matter. As such, forgoing a per-instruction-type text introduction to what the processor does for each instruction and how they relate to each other in the pipeline, the .drawio file provided with the project files lays out a step-by-step extension-process of how to go from the single-cycle version of the processor to the fully functional pipelined version. Each tab of the .drawio file is aptly named to refer to the motivation and purpose behind the extension (note: Ctrl+Shift+H to centre the design in the window and thus see each transition from a sort fo “framed” fixed position). The single-cycle version corresponds to the datapath.vhd file (simulated with the datapath_tb.vhd file) and the pipelined version resides in the datapath_PP{,_tb}.vhd files.

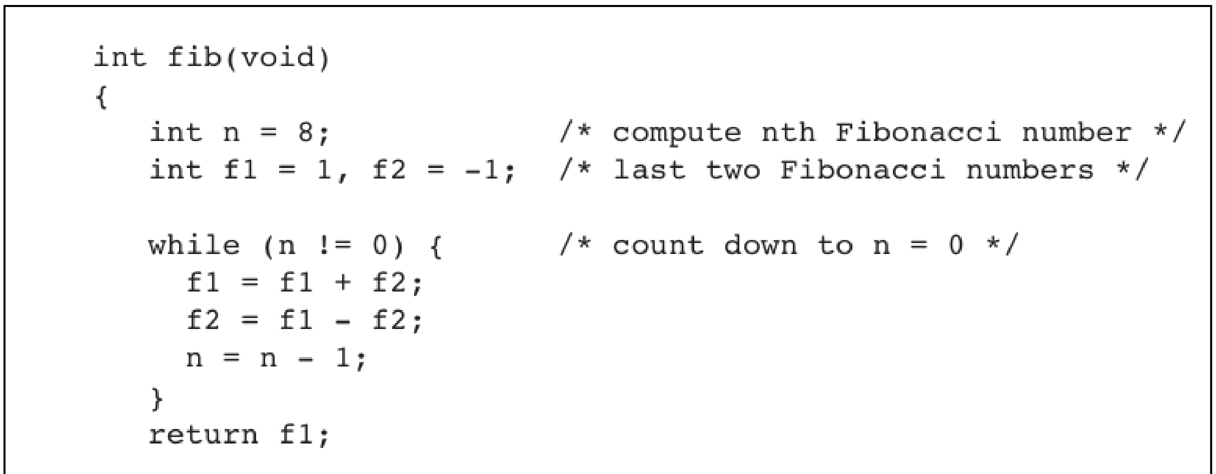




Performance test results

Four performance tests are considered in this section: the n-th Fibonacci calculator, the proposed buffer-sum process, the proposed mem_copy process and the proposed matrix multiplication process. Below are the figures for the original code and their implementation in assembly.

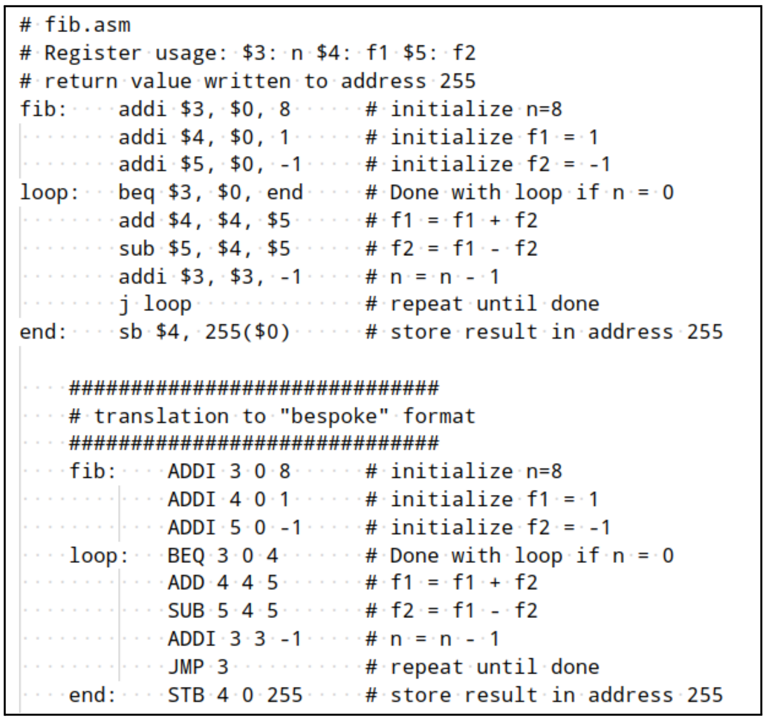
Fibonacci C code



```c
int fib(void)
{
   int n = 8;             /* compute nth Fibonacci number */
   int f1 = 1, f2 = -1;   /* last two Fibonacci numbers */

   while (n != 0) {       /* count down to n = 0 */
     f1 = f1 + f2;
     f2 = f1 - f2;
     n = n - 1;
   }
   return f1;
```

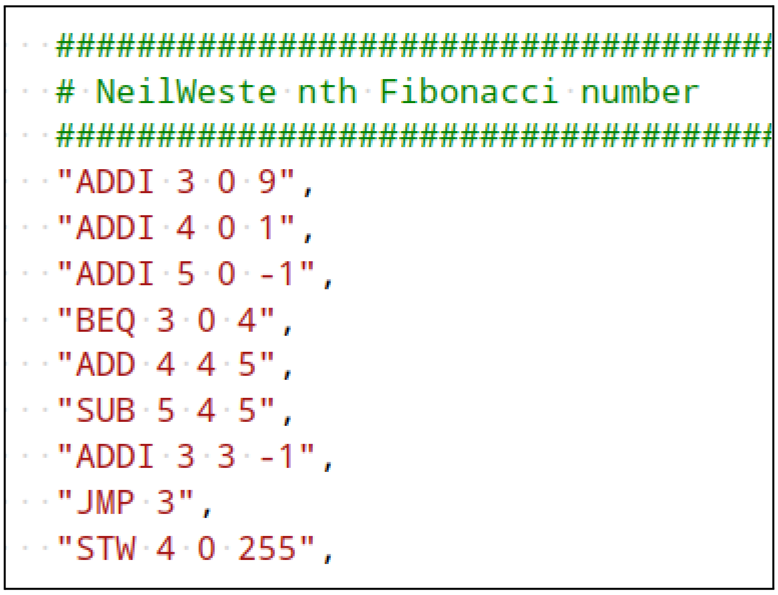
FIGURE 1.50 C Code for Fibonacci program

Fibonacci code in its original (from the book) assembly format and the processor-adapted assembly code for our machine (below)



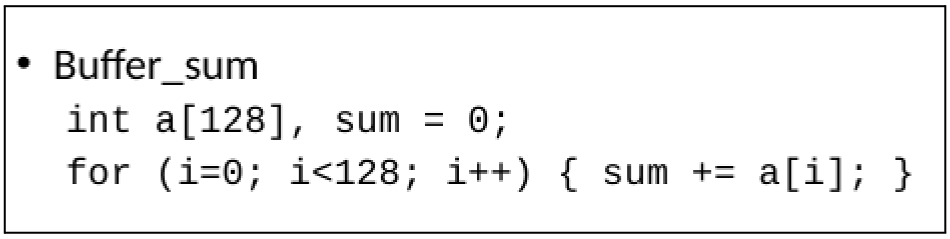
```
# fib.asm
# Register usage: $3: n $4: f1 $5: f2
# return value written to address 255
fib:    addi $3, $0, 8     # initialize n=8
        addi $4, $0, 1     # initialize f1 = 1
        addi $5, $0, -1    # initialize f2 = -1
loop:   beq $3, $0, end    # Done with loop if n = 0
        add $4, $4, $5     # f1 = f1 + f2
        sub $5, $4, $5     # f2 = f1 - f2
        addi $3, $3, -1    # n = n - 1
        j loop             # repeat until done
end:    sb $4, 255($0)     # store result in address 255

    ##############################
    # translation to "bespoke" format
    ##############################
    fib:    ADDI 3 0 8     # initialize n=8
            ADDI 4 0 1     # initialize f1 = 1
            ADDI 5 0 -1    # initialize f2 = -1
    loop:   BEQ 3 0 4      # Done with loop if n = 0
            ADD 4 4 5      # f1 = f1 + f2
            SUB 5 4 5      # f2 = f1 - f2
            ADDI 3 3 -1    # n = n - 1
            JMP 3          # repeat until done
    end:    STB 4 0 255    # store result in address 255
```

Actual assembly code in the asmtoisa.py script



```
    ####################################
    # NeilWeste nth Fibonacci number
    ####################################
    "ADDI 3 0 9",
    "ADDI 4 0 1",
    "ADDI 5 0 -1",
    "BEQ 3 0 4",
    "ADD 4 4 5",
    "SUB 5 4 5",
    "ADDI 3 3 -1",
    "JMP 3",
    "STW 4 0 255",
```

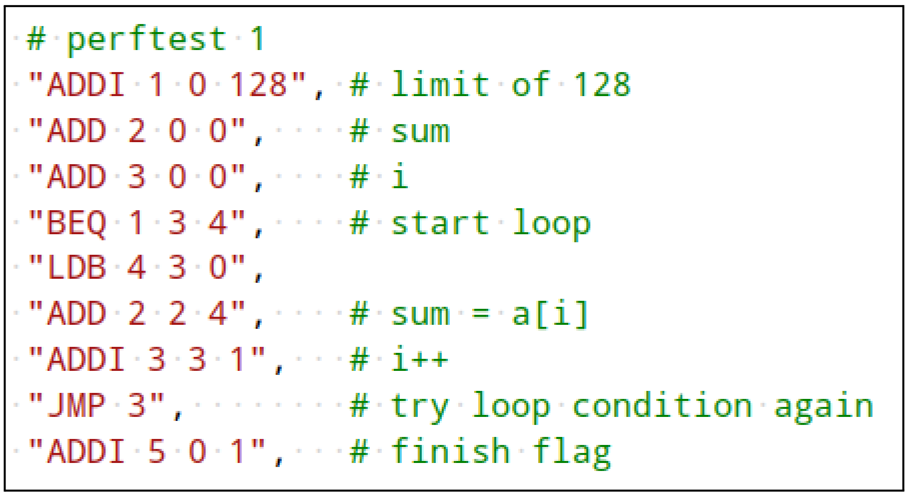
Buffer-sum original code



- Buffer_sum

```
int a[128], sum = 0;
for (i=0; i<128; i++) { sum += a[i]; }
```

Buffer-sum assembly implementation



```
# perftest 1
"ADDI 1 0 128", # limit of 128
"ADD 2 0 0",    # sum
"ADD 3 0 0",    # i
"BEQ 1 3 4",    # start loop
"LDB 4 3 0",
"ADD 2 2 4",    # sum = a[i]
"ADDI 3 3 1",   # i++
"JMP 3",        # try loop condition again
"ADDI 5 0 1",   # finish flag
```

Mem-copy original code

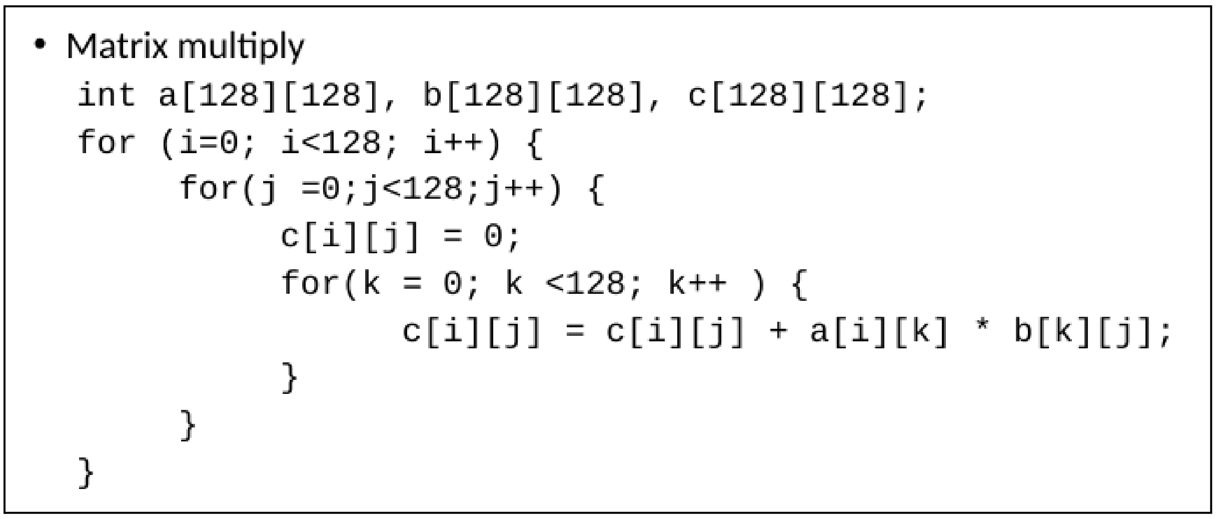
- Mem_copy

```
int a[128], b[128];
for (i=0; i<128; i++) { a[i] = 5; }
for (i=0; i<128; i++) { b[i] = a[i]; }
```

Mem-copy assembly implementation

```
# perftest 2
"ADDI 1 0 128", # limit of 128
"ADDI 2 0 5",   # 5 value
"ADD 3 0 0",    # i
"ADDI 4 0 0",   # base pointer for a[128]
"ADDI 5 0 128", # base pointer for b[128]
"BEQ 1 3 4",    # start loop #1
"STB 2 4 0",    # a[i] = 5
"ADDI 4 4 1",   # update pointer of a[128]
"ADDI 3 3 1",   # i++
"JMP 5",        # try loop condition again
"SUB 4 4 3",    # reset pointer of a[128]
"ADDI 3 0 0",   # reset i
"BEQ 1 3 6",    # start loop #2
"LDB 6 4 0",    # intermediate register to hold a[i] value
"STB 6 5 0",    # b[i] = a[i]
"ADDI 4 4 1",   # update pointer of a[128]
"ADDI 5 5 1",   # update pointer of b[128]
"ADDI 3 3 1",   # i++
"JMP 12",       # try loop condition again
"ADDI 31 0 1",  # finish flag
```

Matrix multiply original code



- Matrix multiply

```
int a[128][128], b[128][128], c[128][128];
for (i=0; i<128; i++) {
     for(j =0;j<128;j++) {
          c[i][j] = 0;
          for(k = 0; k <128; k++ ) {
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
          }
     }
}
```

Matrix multiplication assembly implementation

```
  # perftest 3
  "ADDI 1 0 10", # limit of 128
  "ADD 2 0 0",    # i
  "ADD 3 0 0",    # j
  "ADD 4 0 0",    # k
  "ADDI 5 0 0",   # base pointer for a[][]
  "ADDI 6 0 0",   # base pointer for b[][]
  "ADDI 7 0 0",   # base pointer for c[][]
  "BEQ 1 2 22",   # start loop #1
  "BEQ 1 3 18",   # start loop #2
  "MUL 10 2 1",   # i*128
  "ADD 11 10 3",  # (i*128)+j
  "ADD 12 11 7",  # (i*128)+j+base_c_pointer
  "STB 0 12 0",   # c[i][j] = 0
  "BEQ 1 4 10",   # start loop #3
  # a[i][k] section
  "MUL 13 2 1",   # i*128
  "ADD 14 13 4",  # (i*128)+k
  "ADD 15 14 5",  # (i*128)+k+base_a_pointer
  # b[k][j] section
  "MUL 16 2 1",   # k*128
  "ADD 17 16 3",  # (k*128)+j
  "ADD 18 17 6",  # (k*128)+j+base_b_pointer
  # a[i][k]*b[k][j] section
  "MUL 19 15 18",
  # c[i][j]=c[i][j]+a[i][k]*b[k][j] section
  "ADD 12 12 19",
  "ADDI 4 4 1",   # k++
  "JMP 13",       # try loop #3 condition again
  "ADDI 4 0 0",   # reset k
  "ADDI 3 3 1",   # j++
  "JMP 8",        # try loop #2 condition again
  "ADDI 3 0 0",   # reset j
  "ADDI 2 2 1",   # i++
  "JMP 7",        # try loop #1 condition again
  "ADDI 2 0 0",   # reset i
  "ADDI 8 0 1",   # finish flag
```

Results:

	expected value (single cycle output)	single-cycle #cycles	pipelined #cycles	single-to-pp ratio	pp ipc
fib 10	34	551	691	1.254083485	0.7973950796
fib 50	3483774753	2551	3091	1.211681693	0.8252992559
fib 100	3405478146	5051	6091	1.205899822	0.8292562798
fib 200	305885837	10051	12091	1.202964879	0.8312794641
fib 1000	4057268386	50051	60091	1.200595393	0.8329200712

	single-cycle #cycles	pipelined #cycles	single-to-pp ratio	pp ipc
buffer-sum	6451	9071	1.406138583	0.7111674567
mem-copy	15461	19371	1.252894379	0.7981518765
mx-multiply (i=10)	119601	143051	1.196068595	0.8360724497