THUMBSIM – A single-cycle Thumb simulator

Mike Spivey

1. Thumbsim. This program is a register-level simulator for a partial implementation of the Thumb instruction set. The main things missing are multi-word load and store instructions (including `push` and `pop`), loads and stores for bytes and half-words, and the whole exception mechanism: in short, anything that requires microcode and does not execute in a single cycle. The dreaded Thumb bit is also entirely missing from this simulation. The implementation executes each instruction in a single cycle, with no pipelining: it bears no relation to any actual implementation of the instruction set in hardware.

The simulator can load and execute binary machine code prepared with the standard assembler, providing unimplemented instructions are avoided. The main obstacle to idiomatic programming is that the machine is unable to save the return address of a subroutine to memory without moving it from `LR` to a low register first.

The program is presented using Don Knuth's idea of 'literate programming': parts of the program appear in chunks with a ⟨Symbolic name⟩, and the text that is seen by the C compiler is obtained by beginning with this first, unnamed chunk, then systematically replacing chunk names by their contents until no more names remain. Some chunks, like ⟨ Datapath components 5 ⟩, are defined in several sections: their contents are obtained by concatenating all those sections together. This scheme allows the program to be split into parts with more freedom than if parts had to be whole subroutines, and allows the parts to be presented in a good order for understanding, rather than an order imposed by the compiler. The most vital point is that each chunk of code can be preceded by a lengthy commentary (like the one you are now reading) that explains the code. Two programs prepare the source code for different purposes: `CTANGLE` performs the rearrangement explained above, producing a C program that can be compiled, while `CWEAVE` formats the program so that it can be typeset with TEX.

We'll start by including the standard header files we need. The rest of the program consists of named chunks that we will subsequently define.

```
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <stdint.h>
```

⟨ Global definitions 2 ⟩
⟨ Enumerations for signals 6 ⟩
⟨ Control components 7 ⟩
⟨ Datapath components 5 ⟩
⟨ Instruction decoding 28 ⟩
⟨ Instruction simulator 34 ⟩
⟨ Main program 42 ⟩

2. It's good to begin by defining some basic types: a **Word** is a 32-bit quantity that is usually treated as unsigned, and **Halfword** is a 16-bit integer, big enough to contain one machine instruction. The type **Flags** will be used to hold the four flags $NZCV$, with four bits to spare. And **Bool** is the usual Boolean type.

⟨ Global definitions 2 ⟩ ≡

```
typedef uint32_t Word;
typedef uint16_t Halfword;
typedef uint8_t Flags;
typedef enum {
  false, true
} Bool;
```

See also sections 3 and 48.

This code is used in section 1.

3. The global variables of the program correspond to the state of the machine: there is an array of 16 registers (with the stack pointer `SP`, the link register `LR`, and the program counter `PC` among them), a set of flag bits, and an array of memory words. The memory size is specified here in bytes, but the memory itself is declared as an array of 32-bit words. Few, if any, test programs will need so much as 16kB of memory, but I've made the memory large enough that we can test that long branches work correctly.

```
#define SP 13
#define LR 14
#define PC 15
#define MEMSIZE 16384
⟨ Global definitions 2 ⟩ +≡
  Word regfile[16];
  Flags flags;
  Word mem[MEMSIZE ≫ 2];
```

4. We are going to need some operations that treat words bit-wise. From force of habit, these are defined here as macros: since the sizes and offsets are invariably constants, this gives maximum opportunity for even a simple compiler to do constant folding and simplification.

A bit of TEX magic lets us introduce special notation for selecting bits and bit-fields from integers, and implement the operations as macros. The quantity $x\langle i\rangle = bit(x,i)$ is the i'th bit of x, counting from the least-significant end and numbering from zero: this is a special case of the notation $x\langle j{:}i\rangle = field(x,j,i)$, which denotes the field that contains bits i up to j of x, inclusive.

The sign bit $signbit(x)$ is simply $x\langle 31\rangle$, and $signext(x,n)$, the result of sign-extending an n-bit quantity x, is obtained by copying bit $n-1$ of x into all positions to the left of it: as usual, we can achieve this by treating x as signed and shifting first to the left and then to the right again by $32-n$ bits.

The final group of macros provide facilities for packing and unpacking the status bits $NZCV$.

```
format bit TeX
format field TeX
#define x⟨i⟩ x⟨i:i⟩
#define x⟨j:i⟩ (((x) & ~((Word) ~1 ≪ (j))) ≫ (i))
#define signbit(x) x⟨31⟩
#define signext(n,x) ((Word)((((int32_t)(x)) ≪ (32 - (n))) ≫ (32 - (n))))
#define pack(n,z,c,v) ((n) ≪ 3 | (z) ≪ 2 | (c) ≪ 1 | (v))
#define nbit(f) f⟨3⟩
#define zbit(f) f⟨2⟩
#define cbit(f) f⟨1⟩
#define vbit(f) f⟨0⟩
```

5. Arithmetic-logic unit. Let's begin to build up a collection of architectural components, starting with the ALU and the circuitry that controls it. Many of the functions of the ALU can be implemented in terms of a 32-bit adder with explicit carry-in and carry-out connections. Sadly, C – like most high-level languages – gives us no access to the carries, even if they are available in the underlying machine. For the carry-in, we can simply perform an extra addition. For the carry-out, we could ask to perform the addition in 64 bits, and discard all but 33 bits of the result. It's possible, however, to reconstruct both the carry and the overflow bit by examining the signs of the two operands a and b and of the result r, according to this table:

$signbit(a)$	$signbit(b)$	$signbit(r)$	$cflag$	$vflag$
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	0
1	0	1	0	0
1	1	0	1	1
1	1	1	1	0

Happily, these results can be computed with the compact formulas shown below; for present purposes, we can leave the C compiler to make whatever job it can of evaluating them efficiently.

```
⟨ Datapath components 5 ⟩ ≡
  static Word adder(Word a, Word b, Bool cin, Bool *cflag, Bool *vflag) {
    Word r = a + b + cin;
    *cflag = (signbit(a) + signbit(b) > signbit(r));
    *vflag = (signbit(r) ≠ signbit(a) ∧ signbit(r) ≠ signbit(b));
    return r;
  }
```

See also sections 8, 13, 15, 16, 17, 23, and 25.

This code is used in section 1.

6. By introducing an enumerated type for ALU functions, we avoid any need to concern ourselves with the actual numeric values of the control signals: a hardware designer filling in the details of our high-level design should be free to reassign the values of these internal signals if it helps produce simpler hardware. In addition to the fixed functions *Add*, *Sub*, *And*, etc., there are two extra values *Bit7* and *Bit9* that will be eliminated as part of the decoding process, denoting operations that are either *Add* or *Sub* depending on a bit in the instruction.

```
⟨ Enumerations for signals 6 ⟩ ≡
  typedef enum {
    Add, Sub, And, Eor, Adc, Sbc, Neg, Orr, Mul, Mov, Mvn, Bic, Adr, Bit7, Bit9
  } AluOp;
```

See also sections 12, 14, 20, 22, 24, and 26.

This code is used in section 1.

7. Here's the function *alusel* that fixes the function to be performed before it is fed to the ALU. The operation *op* will come from a decoding ROM, and the extra two values allow the operation that is performed to be partially determined by instruction bits that are not part of the opcode decoded by the ROM.

```
⟨ Control components 7 ⟩ ≡
  static AluOp alusel(AluOp op, Halfword instr) {
    switch (op) {
    case Bit7: return (instr⟨7⟩ ? Sub : Add);
    case Bit9: return (instr⟨9⟩ ? Sub : Add);
    default: return op;
    }
  }
```

See also sections 21 and 27.

This code is used in section 1.

8. The ALU has a control input for the operation, and four data inputs. Two inputs are the two arguments to the operation, but also supplied are the *C* flag from the previous instruction (used by the *Adc* and *Sbc* operations), and the carry bit *shc* computed by the shifter. For some operations, *shc* becomes the carry bit for the whole operation. The outputs are the result of the operation, together with the four flag bits. The *N* and *Z* flags always have the same meaning, and the *C* and *V* flags have meanings dependent on the operation.

There is an infidelity in this implementation, in that any instruction that writes some of the flag bits writes all of them, except that shifts do not write *C* flag if the shift amount is zero.

```
⟨ Datapath components 5 ⟩ +≡
  static Word alu(AluOp op, Word in1, Word in2, Bool cin, Bool shc, Flags *flags)
  {
    Word result;
    Bool cflag = false;
    Bool vflag = false;
    switch (op) {
      ⟨ ALU cases for arithmetic operations 9 ⟩
      ⟨ ALU cases for logical operations 10 ⟩
      ⟨ ALU cases for moves and shifts 11 ⟩
    default: panic("Bad ALU operation %d", op);
    }
    Bool nflag = signbit(result);
    Bool zflag = (result ≡ 0);
    *flags = pack(nflag, zflag, cflag, vflag);
    return result;
  }
```

9. Many of the ALU operations (*Add*, *Sub*, *Adc*, *Sbc*, *Neg*) are implemented using the adder, which itself provides the V and C flags. Also included here are *Mul* and the operation *Adr*, a form of addition that is implicit in the `add pc` and `ldr pc` instructions with `PC`-relative addressing. These instructions are defined to round down the `PC` value (which is aligned to a 2-byte boundary) to make it a multiple of 4. No doubt this operation could be implemented with the same adder, but it is written directly here. The flags don't matter, because neither instruction saves them.

```
⟨ ALU cases for arithmetic operations 9 ⟩ ≡
case Add: result = adder(in1, in2, false, &cflag, &vflag); break;
case Sub: result = adder(in1, ~in2, true, &cflag, &vflag); break;
case Adc: result = adder(in1, in2, cin, &cflag, &vflag); break;
case Sbc: result = adder(in1, ~in2, cin, &cflag, &vflag); break;
case Neg: result = adder(0, ~in2, true, &cflag, &vflag); break;
case Mul: result = in1 * in2; break;
case Adr: result = (in1 + in2) & ~0x3; break;
```

This code is used in section 8.

10. Other operations (*And*, *Eor*, *Orr*, *Bic*) are implemented by means of bitwise Boolean operations, and give V and C flags that are zero.

```
⟨ ALU cases for logical operations 10 ⟩ ≡
case And: result = in1 & in2; break;
case Eor: result = in1 ⊕ in2; break;
case Orr: result = in1 | in2; break;
case Bic: result = in1 & ~in2; break;
```

This code is used in section 8.

11. Two ALU operations simply copy the second input to the output, negating it bitwise in the case of *Mvn*. The *Mov* operation are used in shift instructions to copy the output of the barrel shifter, and there a special rule about the carry bit applies: the last bit shifted out forms the *shc* output of the shifter, and it is copied as the carry output of the ALU. On bigger ARM chips, the same rule applies to the `mvn` instruction, so we follow that convention here.

```
⟨ ALU cases for moves and shifts 11 ⟩ ≡
case Mov: result = in2;
  cflag = shc; break;
case Mvn: result = ~in2;
  cflag = shc; break;
```

This code is used in section 8.

12. Barrel shifter. The barrel shifter supports logical and arithmetic shifts and also rotations. Again, we introduce an enumerated type to avoid being explicit about the encoding.

```
⟨ Enumerations for signals 6 ⟩ +≡
  typedef enum {
    Lsl, Lsr, Asr, Ror
  } ShiftOp;
```

13. The function *shifter*(*op*, *x*, *n*, *cflag*) computes the result of a shift applied to *x* and *n*, and also sets *cflag* to the last bit shifted out, leaving it unchanged if $n = 0$. Left and right shifts are easy, though we have to be careful to make sure C gives us an arithmetic shift for *Asr* by inserting appropriate casts, and we must deal appropriately with shifts by 32 bits or more. The *Ror* operation has to be decomposed into two shifts: we hope the C compiler has the gumption to combine them into an `ror` instruction on the host if one exists.

```
⟨ Datapath components 5 ⟩ +≡
  static Word shifter(ShiftOp op, Word x, int n, Bool *cflag) {
    if (n ≡ 0) return x;
    else {
      Word r = 0;
      Bool c = false;
      switch (op) {
      case Lsl: r = (n ≥ 32 ? 0 : x ≪ n);
        c = (n ≥ 33 ? 0 : x⟨32 − n⟩);
        break;
      case Lsr: r = (n ≥ 32 ? 0 : x ≫ n);
        c = (n ≥ 33 ? 0 : x⟨n − 1⟩);
        break;
      case Asr: r = (Word)((int32_t) x ≫ (n ≥ 32 ? 31 : n));
        c = (n ≥ 33 ? x⟨31⟩ : x⟨n − 1⟩);
        break;
      case Ror: r = x ≫ (n & 0x1f) | x ≪ (32 − (n & 0x1f));
        c = x⟨(n − 1⟩ & 0x1f);
        break;
      default: panic("Bad shift op %d", op);
      }
      *cflag = c;
      return r;
    }
  }
```

14. The control hardware must determine not only what operation the shifter performs, but also the distance by which it shifts. Different instructions have different ways of specifying the amount that their second operand should be shifted. Some use an implicit constant – 0, 1, 2, or 12 – while others have a five-bit immediate field (with a special interpretation for right shifts), and still others take the shift amount from the low-order byte of the first register *ra* read by the instruction. We will introduce an enumerated type to list the possibilities.

```
⟨ Enumerations for signals 6 ⟩ +≡
  typedef enum {
    Sh0, Sh1, Sh2, Sh12, ShImm, ShImR, ShReg
  } ShiftSel;
```

15. A function *shiftsel* interprets a value of this type by selecting the appropriate source. For right shifts by a constant (case *ShImR*), the shift amount is interpreted as a number between 1 and 32 inclusive, with 32 encoded as zero; the tricky macro *shfix* decodes this.

```
⟨ Datapath components 5 ⟩ +≡
#define shfix(x)  ((((x) − 1) & 0x1f) + 1)
  static int shiftsel(ShiftSel s, Word ra, Halfword instr) {
    switch (s) {
    case Sh0: return 0;
    case Sh1: return 1;
    case Sh2: return 2;
    case Sh12: return 12;
    case ShImm: return instr⟨10:6⟩;
    case ShImR: return shfix(instr⟨10:6⟩);
    case ShReg: return ra & 0xff;
    default: panic("Bad shift amount %d", s);
    }
  }
```

16. Register file. So far, the functional units we have introduced have been purely combinational, and we have represented them by a single function that computes the outputs of the unit from its inputs. The register file, however, has internal state, and it is described by two functions, *readreg* and *writeregs*, one to access its existing state, and another to establish a new state.

The function *readreg* reads a register value, respecting the convention that the PC reads as PC + 4.

```
⟨ Datapath components 5 ⟩ +≡
  static Word readreg(int i) {
    if (i ≡ PC) return regfile[i] + 4;
    else return regfile[i];
  }
```

17. The subroutine *writeregs* encapsulates the rules for writing new values to the registers; the parameters are the value *result* computed by the current instruction, the address *nextpc* of the next instruction, a Boolean *regwrite* that indicates whether a register (with number *cRegC*) should be written, and a Boolean *cLink* that indicates a branch-and-link instruction. Whether the explicit write happens or not, registers PC and LR and are still updated in a way special to them.

```
⟨ Datapath components 5 ⟩ +≡
  static void writeregs(Word result, Word nextpc, Bool regwrite, int cRegC, Bool cLink) {
    if (regwrite ∧ cRegC < 14) regfile[cRegC] = result;
    ⟨ Update the link register 18 ⟩
    ⟨ Update the program counter 19 ⟩
  }
```

18. Special rules govern updates to the link register and program counter. Both can be written explicitly, and that takes precedence. Otherwise, in a branch-and-link instruction, the link register is written with the address of the following instruction.

```
⟨ Update the link register 18 ⟩ ≡
  if (regwrite ∧ cRegC ≡ LR) regfile[LR] = result;
  else if (cLink) regfile[LR] = nextpc;
```

This code is used in section 17.

19. The program counter, if not explicitly written by an instruction such as a branch, is updated with the address of the next instruction. Values explicitly written to the PC are rounded down to a multiple of two.

```
⟨ Update the program counter 19 ⟩ ≡
  if (regwrite ∧ cRegC ≡ PC) regfile[PC] = result & ∼1;
  else regfile[PC] = nextpc;
```

This code is used in section 17.

20. So much for the internal behaviour of the register file; but we must also describe the rules that determine which specific registers are read or written by an instruction. We introduce an enumerated type *RegSel* for these rules, which sometimes use explicit bits from the instruction, and other times access registers implicit in the opcode. The ARM documentation calls the fields of the instruction *Rd*, *Rn*, *Rm*, etc., but is not totally consistent, perhaps because the names really pertain to the native encoding of instructions, and they are sometimes jiggled a bit in the Thumb encoding. So let's agree to call the instruction fields *Rx*, *Ry*, *Rz* and *Rw*. The fields *Rxx* and *Ryy* are four-bit register numbers that can name high registers, with *Rxx* not contiguous in the instruction.

```
⟨ Enumerations for signals 6 ⟩ +≡
  typedef enum {
    Rx, Ry, Rz, Rw, Rxx, Ryy, Rsp, Rlr, Rpc
  } RegSel;
```

21. The function *regsel* interprets these rules and returns a register number; it could be implemented in hardware by a multiplexer, with some inputs coming from instruction fields and others wired to constants. This part of the design is rendered more complicated by the large variety of different instruction formats that appear in Thumb code.

```
⟨ Control components 7 ⟩ +≡
  static int regsel(RegSel s, Halfword instr) {
    switch (s) {
    case Rx: return instr⟨2:0⟩;
    case Ry: return instr⟨5:3⟩;
    case Rz: return instr⟨8:6⟩;
    case Rw: return instr⟨10:8⟩;
    case Rxx: return instr⟨7⟩ ≪ 3 | instr⟨2:0⟩;
    case Ryy: return instr⟨6:3⟩;
    case Rsp: return SP;
    case Rlr: return LR;
    case Rpc: return PC;
    default: panic("Bad register specifier %d", s);
    }
  }
```

22. Conditional execution. The Thumb instruction set provides 14 different conditions for branching on the flags, and we need a straightforward combinational circuit to establish their meanings. The numbers from 0 to 13 appear in a field of conditional branch instructions, so it's best to be explicit about the encoding, rather than letting it be determined automatically by the enumerated type.

```
⟨ Enumerations for signals 6 ⟩ +≡
  typedef enum {
    CondEq = 0, CondNe = 1, CondCs = 2, CondCc = 3, CondMi = 4, CondPl = 5,
    CondVs = 6, CondVc = 7, CondHi = 8, CondLs = 9, CondGe = 10, CondLt = 11,
    CondGt = 12, CondLe = 13, CondAl = 14, CondNv = 15
  } Cond;
```

23. Condition codes 14 (always) and 15 (never) are unused by the Thumb instruction set, but they can occur in the relevant bit-positions of instructions that are not branches, so we must do *something* with them, or unexpected crashes will result. (In native ARM code, unconditional instructions have a condition code of 14, and 15 is the natural complement of that.)

```
⟨ Datapath components 5 ⟩ +≡
  static Bool condition(Cond cond, Flags flags) {
    Bool n = nbit(flags), z = zbit(flags), v = vbit(flags), c = cbit(flags);
    switch (cond) {
    case CondEq: return z;
    case CondNe: return ¬z;
    case CondCs: return c;
    case CondCc: return ¬c;
    case CondMi: return n;
    case CondPl: return ¬n;
    case CondVs: return v;
    case CondVc: return ¬v;
    case CondHi: return (c ∧ ¬z);
    case CondLs: return (¬c ∨ z);
    case CondGe: return (n ≡ v);
    case CondLt: return (n ≠ v);
    case CondGt: return (¬z ∧ n ≡ v);
    case CondLe: return (z ∨ n ≠ v);
    case CondAl: return true;
    case CondNv: return false;
    default: panic("Bad condition code %d", cond);
    }
  }
```

24. Other control modules. There are a few more details that must be settled before we are ready to put the whole simulation together. There is a multiplexer that selects the value *rand2* fed as input to the barrel shifter. Sometimes this is the second register *rb* read by the instruction, but it can also be drawn from signed (like *SImm8*) or unsigned (like *Imm8*) immediate fields in the instruction of various sizes and locations.

```
⟨ Enumerations for signals 6 ⟩ +≡
  typedef enum {
    RegB, RImm3, Imm5, Imm7, Imm8, SImm8, Imm11, SIm11
  } Rand2Sel;
```

25. Again, there is a function that interprets the control signal by selecting the register value *rb*, or an appropriate field of the instruction and sign-extending it when necessary. We'll label it as a datapath component this time, because its output is determined dynamically if it comes from a register.

```
⟨ Datapath components 5 ⟩ +≡
  static Word rand2sel(Rand2Sel s, Word rb, Halfword instr) {
    switch (s) {
    case RegB: return rb;
    case RImm3: return (instr⟨10⟩ ? instr⟨8:6⟩ : rb);
    case Imm5: return instr⟨10:6⟩;
    case Imm7: return instr⟨6:0⟩;
    case Imm8: return instr⟨7:0⟩;
    case SImm8: return signext(8, instr⟨7:0⟩);
    case Imm11: return instr⟨10:0⟩;
    case SIm11: return signext(11, instr⟨10:0⟩);
    default: panic("Bad rand2 code %d", s);
    }
  }
```

26. A couple of control signals in the machine can be set as definitely true or false for some instruction, or conditional on some other signal. The type **Perhaps** gives a convenient way of representing them.

⟨ Enumerations for signals 6 ⟩ +≡
```
typedef enum {
  yes, no, maybe
} Perhaps;
```

27. When the time comes for a definite answer, the companion function *perhaps* produces one, given the interpretation to be attached to *maybe*. It corresponds to a three-way multiplexer in the hardware.

⟨ Control components 7 ⟩ +≡
```
static Bool perhaps(Perhaps p, Bool c) {
  switch (p) {
  case yes: return true;
  case no: return false;
  case maybe: return c;
  default: panic("Aye, there's the rub!");
  }
}
```

28. Decoding tables. The first step in executing an instruction after it has been fetched is to decode it, producing a bundle of control signals. Decoding an instruction gives a list of 12 control signals, and also a name that is useful for debugging.

- *mnem* represents the mnemonic for the instruction, printed as part of the execution trace.
- *cRegSelA*, *cRegSelB* and *cRegSelC* are *register selectors* that determine how to select the three registers that are read or written by the instruction.
- *cRand2* determines where the second ALU operand comes from.
- *cShiftOp* and *cShiftAmt* determine how that operand is treated by the barrel shifter.
- *sAluSel* determines what ALU operation is performed.
- *cMemRd* and *cMemWr* determine whether a memory read or write happens.
- *cWFlags* determines whether the flags are updated.
- *cWReg* determines whether the result of the instruction is written to a register.
- *cWLink* determines whether the address of the following instruction is written into `lr`.

The *cWReg* and *cWLink* fields have type **Perhaps**, with possible values *yes*, *no* and *maybe*, meaning that the answer will be determined by some other condition. In the case of *cWReg*, conditional branches work by computing the target address regardless of whether the branch is taken or not, but writing it into the `PC` only if the condition is satisfied, and a value of *maybe* here means that the write is conditional. In the case of *cWLink*, there is another bit in the instruction that is not taken into account by the decoder, but makes the difference between `bx` and `blx`.

⟨ Instruction decoding 28 ⟩ ≡
```
typedef struct {
  char *mnem;
  RegSel cRegSelA, cRegSelB, cRegSelC;
  Rand2Sel cRand2;
  ShiftOp cShiftOp;
  ShiftSel cShiftAmt;
  AluOp cAluSel;
  Bool cMemRd, cMemWr, cWFlags;
  Perhaps cWReg, cWLink;
} Control;
```

See also sections 29, 31, 32, and 33.

This code is used in section 1.

29. There are three decoding tables, used for different ranges of opcodes: these could become ROMs in a hardware implementation, or a single PAL that is effectively a ROM with incomplete address decoding. The majority of instructions *instr* can be decoded by looking up the five bits *instr*⟨16:11⟩ in table *decode1*, unless those bits are 01000, in which case we must consult one of two other tables. We don't implement byte and halfword loads and stores, nor `push`, `pop` and adjacent operations, so this is enough. If we did want to add some of these operations, then further auxiliary tables would help with the decoding.

Note that a `cmp` instruction is identical with a `subs` instruction, except that it doesn't write the result back into a register. As indicated earlier, a conditional branch instruction `b<c>` uses the ALU to compute the branch target, and writes it into the `PC` only if the condition is true.

We use *T* and *F* as abbreviations for *true* and *false*, use *Y*, *N* and *C* as abbreviations for the **Perhaps** values, and write _ for don't-care fields and *missing* for instructions that are entirely missing.

```
#define T true
#define F false
#define Y yes
#define N no
#define C maybe
#define _ 0
#define missing {"missing",_,_,_,_,_,_,_,_,_,_,_,_}
⟨ Instruction decoding 28 ⟩ +≡
  const Control decode1[32] = {
  {"lsls",       _,   Ry,  Rx,  RegB,   Lsl,  ShImm, Mov,  F,F,T,Y,N},   /* 0 */
  {"lsrs",       _,   Ry,  Rx,  RegB,   Lsr,  ShImR, Mov,  F,F,T,Y,N},   /* 1 */
  {"asrs",       _,   Ry,  Rx,  RegB,   Asr,  ShImR, Mov,  F,F,T,Y,N},   /* 2 */
  {"adds/subs",  Ry,  Rz,  Rx,  RImm3,  Lsl,  Sh0,   Bit9, F,F,T,Y,N},   /* 3 */
  {"movs i8",    _,   _,   Rw,  Imm8,   Lsl,  Sh0,   Mov,  F,F,T,Y,N},   /* 4 */
  {"cmp i8",     Rw,  _,   _,   Imm8,   Lsl,  Sh0,   Sub,  F,F,T,N,N},   /* 5 */
  {"adds i8",    Rw,  _,   Rw,  Imm8,   Lsl,  Sh0,   Add,  F,F,T,Y,N},   /* 6 */
  {"subs i8",    Rw,  _,   Rw,  Imm8,   Lsl,  Sh0,   Sub,  F,F,T,Y,N},   /* 7 */
  missing,     /* 8: see below */
  {"ldr pc",     Rpc, _,   Rw,  Imm8,   Lsl,  Sh2,   Adr,  T,F,F,Y,N},   /* 9 */
  {"str r",      Ry,  Rz,  Rx,  RegB,   Lsl,  Sh0,   Add,  F,T,F,N,N},   /* 10 */
  {"ldr r",      Ry,  Rz,  Rx,  RegB,   Lsl,  Sh0,   Add,  T,F,F,Y,N},   /* 11 */
  {"str i5",     Ry,  _,   Rx,  Imm5,   Lsl,  Sh2,   Add,  F,T,F,N,N},   /* 12 */
  {"ldr i5",     Ry,  _,   Rx,  Imm5,   Lsl,  Sh2,   Add,  T,F,F,Y,N},   /* 13 */
  missing,     /* 14: Only whole-word loads and stores */
  missing,     /* 15 */
  missing,     /* 16 */
  missing,     /* 17 */
  {"str sp",     Rsp, _,   Rw,  Imm8,   Lsl,  Sh2,   Add,  F,T,F,N,N},   /* 18 */
  {"ldr sp",     Rsp, _,   Rw,  Imm8,   Lsl,  Sh2,   Add,  T,F,F,Y,N},   /* 19 */
  {"add pc",     Rpc, _,   Rw,  Imm8,   Lsl,  Sh2,   Adr,  F,F,F,Y,N},   /* 20 */
  {"add sp",     Rsp, _,   Rw,  Imm8,   Lsl,  Sh2,   Add,  F,F,F,Y,N},   /* 21 */
  {"add/sub sp", Rsp, _,   Rsp, Imm7,   Lsl,  Sh2,   Bit7, F,F,F,Y,N},   /* 22 */
  missing,     /* 23: No push or pop */
  missing,     /* 24: No stm */
  missing,     /* 25: No ldm */
  {"b<c>",       Rpc, _,   Rpc, SImm8,  Lsl,  Sh1,   Add,  F,F,F,C,N},   /* 26 */
  {"b<c>",       Rpc, _,   Rpc, SImm8,  Lsl,  Sh1,   Add,  F,F,F,C,N},   /* 27 */
  {"b",          Rpc, _,   Rpc, SIm11,  Lsl,  Sh1,   Add,  F,F,F,Y,N},   /* 28 */
  missing,     /* 29: Reserved? */
  ⟨ Decoding for the bl instruction 30 ⟩
  };
```

30. The long `bl` instruction is 32 bits long instead of 16, so as to encode 22 bits of displacement between the instruction address and the address of the subroutine it calls. On early implementations of the architecture, it could be executed as two separate instructions, and that is the approach we follow here. The first half `bl1` adds together the `PC` value and the high-order bits of the displacement and saves the result in `LR`, and the second half `bl2` takes this `LR` value, adds on the low-order bits of the displacement, and writes the result into the `PC`, at the same time storing the return address in `LR`. Later revisions of the architecture have added further significant bits in `bl2`, making it impossible to decode it as an independent instruction.

```
⟨ Decoding for the bl instruction 30 ⟩ ≡
  {"bl1",   Rpc, _,   Rlr, SIm11, Lsl, Sh12, Add, F, F, F, Y, N},   /* 30 */
  {"bl2",   Rlr, _,   Rpc, Imm11, Lsl, Sh1,  Add, F, F, F, Y, Y },  /* 31 */
```

This code is used in section 29.

31. The second table applies to instructions that start with the six bits 010000. These are all arithmetic instructions with the same format; but note that the first and second registers are swapped in those instructions that perform shifts, because the data input to the shifter is taken from *rb* and the shift amount from *ra*. The table is indexed by bits ⟨9:6⟩ of the instruction, assuming bits ⟨15:10⟩ are 010000.

```
⟨ Instruction decoding 28 ⟩ +≡
  const Control decode2[16] = {
  {"ands",  Rx, Ry, Rx, RegB, Lsl, Sh0,   And, F, F, T, Y, N},   /* 0 */
  {"eors",  Rx, Ry, Rx, RegB, Lsl, Sh0,   Eor, F, F, T, Y, N},   /* 1 */
  {"lsls",  Ry, Rx, Rx, RegB, Lsl, ShReg, Mov, F, F, T, Y, N},   /* 2 */
  {"lsrs",  Ry, Rx, Rx, RegB, Lsr, ShReg, Mov, F, F, T, Y, N},   /* 3 */
  {"asrs",  Ry, Rx, Rx, RegB, Asr, ShReg, Mov, F, F, T, Y, N},   /* 4 */
  {"adcs",  Rx, Ry, Rx, RegB, Lsl, Sh0,   Adc, F, F, T, Y, N},   /* 5 */
  {"sbcs",  Rx, Ry, Rx, RegB, Lsl, Sh0,   Sbc, F, F, T, Y, N},   /* 6 */
  {"rors",  Ry, Rx, Rx, RegB, Ror, ShReg, Mov, F, F, T, Y, N},   /* 7 */
  {"tst",   Rx, Ry, _,  RegB, Lsl, Sh0,   And, F, F, T, N, N},   /* 8 */
  {"negs",  Rx, Ry, Rx, RegB, Lsl, Sh0,   Neg, F, F, T, Y, N},   /* 9 */
  {"cmp",   Rx, Ry, _,  RegB, Lsl, Sh0,   Sub, F, F, T, N, N},   /* 10 */
  {"cmn",   Rx, Ry, _,  RegB, Lsl, Sh0,   Add, F, F, T, N, N},   /* 11 */
  {"orrs",  Rx, Ry, Rx, RegB, Lsl, Sh0,   Orr, F, F, T, Y, N},   /* 12 */
  {"muls",  Rx, Ry, Rx, RegB, Lsl, Sh0,   Mul, F, F, T, Y, N},   /* 13 */
  {"bics",  Rx, Ry, Rx, RegB, Lsl, Sh0,   Bic, F, F, T, Y, N},   /* 14 */
  {"mvns",  Rx, Ry, Rx, RegB, Lsl, Sh0,   Mvn, F, F, T, Y, N},   /* 15 */
  };
```

32. The third table applies to instructions that start with 010001. They all allow access to all 16 registers, unlike other instructions that can address only the lower eight. This table is indexed by bits ⟨9:8⟩ of the instruction, assuming bits ⟨15:10⟩ are 010001.

```
⟨ Instruction decoding 28 ⟩ +≡
  const Control decode3[4] = {
  {"add hi",  Rxx, Ryy, Rxx, RegB, Lsl, Sh0,   Add, F, F, F, Y, N},   /* 0 */
  {"cmp hi",  Rxx, Ryy, _,   RegB, Lsl, Sh0,   Sub, F, F, T, N, N},   /* 1 */
  {"mov hi",  _,   Ryy, Rxx, RegB, Lsl, Sh0,   Mov, F, F, F, Y, N},   /* 2 */
  {"bx/blx",  _,   Ryy, Rpc, RegB, Lsl, Sh0,   Mov, F, F, F, Y, C},   /* 3 */
  };
```

33. We select the table that is used by looking at the 5-bit opcode of the instruction, treating opcode 8 as a special case. The function *decode* does the job, returning a pointer to the appropriate record of control signals.

```
⟨ Instruction decoding 28 ⟩ +≡
  static const Control *decode(Halfword instr) {
    int op = instr⟨15:11⟩;
    if (op ≠ 8) return &decode1[op];
    else if (¬instr⟨10⟩) return &decode2[instr⟨9:6⟩];
    else return &decode3[instr⟨9:8⟩];
  }
```

34. Executing instructions. The heart of the simulator is the function *step* that takes a processor state and from it computes the next state that will exist one clock cycle later. The work of simulating an instruction is divided into seven stages that we will set out in turn. For faithfulness to the nature of hardware, we perform all seven of these stages for each instruction, even if it does not need to use their results. For example, every instruction has a memory access phase, though if the *memRd* and *memWr* controls are false, it will do nothing. More seriously, bits ⟨11:8⟩ of every instruction will be interpreted as a condition and evaluated against the flags whether the instruction is a conditional branch or not, even though the result may be nonsense: that's why the function *condition* defined above had to be made total, and not allowed to fail when the instruction field contains 14 or 15.

This approach – with decoding ROMs and fixed stages – is not the best if we simply want a true software *emulator* for the machine that allows its code to be run on another platform. For that purpose, it would be better to have a 'big switch' on the opcode (with nested switches where we have multiple ROMs), with each arm of the switch containing specialised code for one instruction. Our approach is oriented more towards demonstrating the validity of a hardware design than to maximising emulated execution speed.

```
⟨ Instruction simulator 34 ⟩ ≡
  static void step(void) {
    ⟨ Fetch and decode an instruction 35 ⟩
    printf("      %s\n", ctrl→mnem);
    ⟨ Compute derived control signals 36 ⟩
    ⟨ Read registers ra, rb, rc 37 ⟩
    ⟨ Compute the shifter output aluin2 38 ⟩
    ⟨ Perform the ALU function and compute newflags 39 ⟩
    ⟨ Access the memory and set result 40 ⟩
    ⟨ Conditionally write back result and newflags 41 ⟩
  }
```

This code is used in section 1.

35. The first job is to fetch and decode an instruction. For simplicity, we adopt a 'modified Harvard architecture', where the single memory of the machine is presented via two interfaces that can be thought of as independent caches. This simulation does not include cache misses, so in effect we have a two-port memory, capable of fetching an instruction word and either reading or writing a data word in each cycle. Real implementations of the instruction set are not like this, having either a single cache or no cache at all, and inserting an extra cycle for load and store operations via a single interface to memory.

The memory is organised in 4-byte words, so we need to fetch a word here and select one half or the other. Bit 1 of the `PC` identifies which half, and bit 0 is ignored.

```
⟨ Fetch and decode an instruction 35 ⟩ ≡
  Word pc = regfile[PC];
  if (pc > MEMSIZE) panic("PC out of range at %u", pc);
  Halfword instr = (pc⟨1⟩ ? mem[pc ≫ 2]⟨31:16⟩ : mem[pc ≫ 2]⟨15:0⟩);
  const Control *ctrl = decode(instr);
  Word nextpc = pc + 2;
```

This code is used in section 34.

36. Next we compute some derived control signals, most of them consisting of bit-fields selected from the instruction, with the format determined by the earlier parts of the decoding process. All this is made more complicated in the Thumb architecture than in some other RISC designs because of the great variety of instruction formats.

It's good to distinguish between the decoded signals (those that appear in the decoding tables and are determined by the instruction's opcode), the derived signals that also depend on other parts of the instruction halfword, and dynamic signals that depend on other parts of the machine state.

- The *decoded* signals will be the same in every instance of an instruction – that it to say, two instructions that share the same opcode will have the same decoded signals. Note, however, that two instructions like "`mov r`" and "`mov i8`" may share the same mnemonic in assembly language, but have different opcodes, and so be treated as different instructions by the hardware, with different decoded signals. In the simulator, these decoded signals form the members of the **Control** structure *ctrl* contained in one of the decoding tables, and have names like *ctrl*→*cShiftAmt*.
- The *derived* signals will be the same whenever a particular instruction is executed, because they are determined by the bits of the instruction taken all together; for example, the instruction

  ```
  add r3, r1, r2
  ```

  always writes its result to register `r3`, and so has *cRegC* = 3. In the simulator, these derived signals are outside the **Control** structure, but also have names like *cRegC* that start with a lower-case *c* and contain capitals.
- The *dynamic* signals differ from one execution of the instruction to another, so that at one time that `add` instruction could write 7 to `r3`, and another time it could write 8, and the value of the *result* signal would be different in the two cases.

⟨ Compute derived control signals 36 ⟩ ≡
```
int cRegA = regsel(ctrl→cRegSelA, instr);
int cRegB = regsel(ctrl→cRegSelB, instr);
int cRegC = regsel(ctrl→cRegSelC, instr);
AluOp cAluOp = alusel(ctrl→cAluSel, instr);
Cond cCond = instr⟨11:8⟩;
Bool cLink = perhaps(ctrl→cWLink, instr⟨7⟩);
```
This code is used in section 34.

37. The next step is to read the three registers that have been selected. In the hardware, these registers are read whether their values are needed or not, and indeed whether the register numbers selecting them make any sense or not. We do the same here.

It's an instruction like `str r1, [r2, r3]` that shows why the datapath must be able to read three registers; in this instruction `r2` and `r3` are read to give the values *ra* and *rb*. The ALU adds these together, and then the value *rc*, read from `r1`, is stored there.

⟨ Read registers *ra*, *rb*, *rc* 37 ⟩ ≡
```
Word ra = readreg(cRegA);
Word rb = readreg(cRegB);
Word rc = readreg(cRegC);
```
This code is used in section 34.

38. The input to the barrel shifter is either the second register *rb* that was read, or an immediate field from the instruction, according to the control signal *ctrl*→*cRand2*. We use the shifter to multiply such immediate fields by a power of two when they form part of an address. The shift amount may be implicit in the instruction, may be specified by an immediate field, or may be the bottom five bits of *ra*, the first register read by the instruction.

⟨ Compute the shifter output *aluin2* 38 ⟩ ≡
```
Word shiftin = rand2sel(ctrl→cRand2, rb, instr);
int shiftamt = shiftsel(ctrl→cShiftAmt, ra, instr);
Bool shc = cbit(flags);
Word aluin2 = shifter(ctrl→cShiftOp, shiftin, shiftamt, &shc);
```
This code is used in section 34.

39. Register value *ra* is always the first input to the ALU, and the output from the shifter forms the second input. The ALU gets both the current *C* flag (for use in the `adc` and `sbc` instructions), and the carry flag *shc* output by the shifter, because in shift instructions this last bit shifted out becomes the *C* flag in the result.

⟨ Perform the ALU function and compute *newflags* 39 ⟩ ≡
```
Flags newflags;
Word aluout = alu(cAluOp, ra, aluin2, cbit(flags), shc, &newflags);
```
This code is used in section 34.

40. Memory access is performed only if control signals ask for it: since the address is computed by the ALU, we must be sure to complain about an address out of range only if a memory access is actually being performed. Only word-sized accesses are supported here, and we round down the address to a multiple of 4. The value written in a store instruction is *rc*, the third register read from the register file earlier. A load instruction takes its result from the memory; otherwise it is the ALU result.

⟨ Access the memory and set *result* 40 ⟩ ≡
```
Word memout = 0;
if (ctrl→cMemRd) {
  if (aluout > MEMSIZE) panic("Memory read out of range at %u", aluout);
  memout = mem[aluout ≫ 2];
}
if (ctrl→cMemWr) {
  if (aluout > MEMSIZE) panic("Memory write out of range at %u", aluout);
  mem[aluout ≫ 2] = rc;
}
Word result = (ctrl→cMemRd ? memout : aluout);
```
This code is used in section 34.

41. The action of writing back the result of the instruction to a register can be made conditional on the flags from the previous instruction. This allows us to implement a conditional branch as a conditional update of the `PC`. The flags are also updated if the instruction requires it.

⟨ Conditionally write back *result* and *newflags* 41 ⟩ ≡
```
Bool enable = condition(cCond, flags);
Bool regwrite = perhaps(ctrl→cWReg, enable);
writeregs(result, nextpc, regwrite, cRegC, cLink);
if (ctrl→cWFlags) flags = newflags;
```
This code is used in section 34.

42. Main program. That concludes the details of the simulation. All that remains is to put together a simple main program to drive it. The program is invoked with arguments that first name a binary image file, then optionally list up to 13 values to be loaded into the first few registers. After loading the image and initialising the registers, the program enters a loop that simulates the Thumb machine cycle by cycle, printing a compact summary of the machine state before executing each instruction.

```
⟨ Main program 42 ⟩ ≡
  int main(int argc, char **argv) {
    if (argc < 2) {
      fprintf(stderr, "Usage: thumbsim binfile r0 r1 ...\n");
      exit(2);
    }
    ⟨ Read the binary image 43 ⟩
    ⟨ Initialise the registers 45 ⟩
    while (true) {
      ⟨ Print the machine state 44 ⟩
      if (⟨ Time to stop? 47 ⟩) break;
      step( );
    }
    printf("exit %u\n", regfile[0]);
    return 0;
  }
```

This code is used in section 1.

43. We will depend on the standard assembler and linker for the Thumb machine to put together a binary file that contains an exact image of the initial contents of the memory, and the main program loads this into the *mem* array before execution starts. This is a simple task, provided we yield to the temptation to read the image as a single block, exploiting the assumption that the machine running the simulation is little-endian like the Thumb architecture. If so, then accessing the unchanged binary image word by word (for example, in the expression $mem[aluout \gg 2]$ that appears in section ⟨ Access the memory and set *result* 40 ⟩ will give the right results.

```
⟨ Read the binary image 43 ⟩ ≡
  FILE *fp = fopen(argv[1], "rb");
  if (fp ≡ Λ) {
    fprintf(stderr, "thumbsim: can't read %s\n", argv[1]);
    exit(1);
  }
  fread(mem, 1, MEMSIZE, fp);
  if (¬feof(fp)) {
    fprintf(stderr, "thumbsim: binary file was too big\n");
    exit(1);
  }
  fclose(fp);
```

This code is used in section 42.

44. We print the machine state in a compact form that shows the PC, four of the registers, and the flags, printing each flag with either its single-letter name or a dot.

```
#define flag(b, ch) (b ? ch : '.')
```

```
⟨ Print the machine state 44 ⟩ ≡
  printf("%04x: %10d %10d %10d %10d  %c%c%c%c\n",
      regfile[PC] & 0xffff, regfile[0], regfile[1], regfile[2], regfile[3],
      flag(nbit(flags), 'N'), flag(zbit(flags), 'Z'), flag(cbit(flags), 'C'), flag(vbit(flags), 'V'));
```

This code is used in section 42.

45. The first few registers (up to 13 of them) are initialised from the command line, and the last three are given specific initial values. Using *strtoul* to convert the argument strings allows hexadecimal constants beginning with 0x as well as decimal ones. Any extra arguments are ignored.

⟨ Initialise the registers 45 ⟩ ≡
```
for (int i = 0; i < 13 ∧ i + 2 < argc; i++)
    regfile[i] = strtoul(argv[i + 2], Λ, 0);
⟨ Set initial values for SP, LR and PC 46 ⟩
```
This code is used in section 42.

46. The stack pointer (register 13) is initialised to the top of memory, and the program counter to 0, with the assumption that the program has been built with 0 as its start address. The initial value of the link register is the magic value `MAGIC` = `0xfffffffe`.

```
#define MAGIC 0xfffffffe
```
⟨ Set initial values for SP, LR and PC 46 ⟩ ≡
```
regfile[SP] = MEMSIZE;
regfile[LR] = MAGIC;
regfile[PC] = 0;
```
This code is used in section 45.

47. The simulation halts if the value `MAGIC` ever appears in the `PC`. This convention allows the simulated program to be written as a subroutine, halting the simulation when the subroutine returns.

⟨ Time to stop? 47 ⟩ ≡
```
regfile[PC] ≡ MAGIC
```
This code is used in section 42.

48. One final detail: the function *panic* stops the simulation abruptly if something bad happens. Hopefully the message that is printed will give a clue to the cause.

```
#define NORETURN __attribute((noreturn))
```
⟨ Global definitions 2 ⟩ +≡
```
static void NORETURN panic(char *msg, ... ) {
    va_list va;
    fflush(stdout);
    fprintf(stderr, "Panic: ");
    va_start(va, msg);
    vfprintf(stderr, msg, va);
    va_end(va);
    fprintf(stderr, "\n");
    exit(2);
}
```

49. The index. This section contains, first, an index showing which numbered sections mention each identifier used in the program. Underlining is used to highlight the sections where each identifer is declared. That index is followed by another, an 'index of first lines' showing the name of each chunk of text, which sections contribute to it, and where it is used.