1 Basic behavior

The microcoded machine (μM) is built around a register bank and an 8-bit ALU with registered operands T1 and T2. It performs all its operations in two cycles, so I have divided it in two stages: an operand stage and an ALU stage. This is nothing more than a 2-stage pipeline.

In the operand stage, registers T1 and T2 are loaded with either the contents of the register bank (RB) or the input signal DI.
In the ALU stage, the ALU output is written back into the RB or loaded into the output register DO. Besides, flags are updated, or not, according to the microinstruction (μI) in execution.

Every microinstruction controls the operation of the operand stage and the succeeding ALU stage; that is, the execution of a μI extends over 2 succeeding clock cycles, and microinstructions overlap each other. This means that the part of the μI that controls the 2nd stage has to be pipelined; in the VHDL code, I have divided the μI in a field_1 and a field_2, the latter of which is registered (pipelined) and controls the 2nd μM stage (ALU).
Many of the control signals are encoded in the microinstructions in what I have improperly called flags. You will see many references to flags in the following text (#end,#decode, etc.). They are just signals that you can activate individually in each μI, some are active in the 1st stage, some in the 2nd. They are all explained in section 3.4.

Note that microinstructions are atomic: both stages are guaranteed to execute in all circumstances. Once the 1st stage of a μI has executed, the only thing that can prevent the execution of the 2nd stage is a reset.
It might have been easier to design the machine so that microinstructions executed in one cycle, thus needing no pipeline for the μI itself. I arbitrarily chose to 'split' the microcode execution, figuring that it would be easier for me to understand and program the microcode; in hindsight it may have been a mistake but in the end, once the debugging is over, it makes little difference.

The core as it is now does not support wait states: it does all its external accesses (memory or i/o, read or write) in one clock cycle. It would not be difficult to improve this with some little modification to the micromachine, without changes to the microcode.
Since the microcode rom is the same type of memory as will be used for program memory, the main advantage of microprogramming is lost. Thus, it would make sense to develop the core a bit further with support for wait states, so it could take advantage of the speed difference between the FPGA and external slow memory.
The register bank reads asynchronously, while writes are synchronous. This is the standard behaviour of a Spartan LUT-based RAM. The register bank holds all the 8080 registers, including the accumulator, plus temporary, 'hidden' registers (x,y,w,z). Only the PSW register is held out of the register bank, in a DFF-8 register.

2 Micromachine control

2.1 Microcode operation

There is little more to the core that what has already been said; all the CPU operations are microcoded, including interrupt response, reset and instruction opcode fetch. The microcode source code can be seen in file `ucode/light8080.m80`, in a format I expect will be less obscure than a plain vhdl constant table.

The microcode table is a synchronous ROM with 512 32-bit words, designed to fit in a Spartan 3 block ram. Each 32-bit word makes up a microinstruction. The microcode 'program counter' (uc_addr in the VHDL code) thus is a 9-bit register.
Out of those 512 words, 256 (the upper half of the table) are used as a jump-table for instruction decoding. Each entry at 256+NN contains a 'JSR' μI to the start of the microcode for the instruction whose opcode is NN. This seemingly unefficient use of RAM is in fact an optimization for the Spartan-3 architecture to which this design is tailored – the 2KB RAM blocks are too large for the microcode so I chose to fill them up with the decoding table.
This scheme is less than efficient where smaller RAM blocks are available (e.g. Altera Stratix).
The jump table is built automatically by the microcode assembler, as explained in section 4.
The upper half of the table can only be used for decoding; JSR instructions can only point to the lower half, and execution from address 0x0ff rolls over to 0x00 (or would; the actual microcode does not use this 'feature').

The ucode address counter uc_addr has a number of possible sources: the micromachine supports one level of micro-subroutine calls; it can also return from those calls; the uc_addr gets loaded with some constant values upon reset, interrupt or instruction fetch. And finally, there is the decoding jump table mentioned above. So, in summary, these are the possible sources of uc_addr each cycle:

- Constant value of 0x0001 at reset (see VHDL source for details).
- Constant value of 0x0003 at the beginning (fetch cycle) of every instruction.
- Constant value of 0x0007 at interrupt acknowledge.
- uc_addr + 1 in normal microinstruction execution
- Some 8-bit value included in JSR microinstructions (calls).
- The return value preserved in the last JSR (used when flag #ret is raised)

All of this is readily apparent, I hope, by inspecting the VHDL source. Note that there is only one jump microinstruction (JSR) which doubles as 'call'; whenever a jump is taken the the 1-level-deep 'return stack' is loaded with the return address (address of the μI following the jump). You just have to ignore the return address when you don't need it (e.g. the jumps in the decoding jump table). I admit this scheme is awkward and inflexible; but it was the first I devised, it works and

fits the area budget: more than enough in this project. A list of all predefined, 'special' microcode addresses follows.

- **0x001 – reset**
  After reset, the μI program counter (uc_addr in the VHDL code) is initialized to 0x00. The program counter works as a pre-increment counter when reading the microcode rom, so the μI at address 0 never gets executed (unless 'rolling over' from address 0x0ff, which the actual microcode does not). Reset starts at address 1 and takes 2 microinstructions to clear PC to 0x0000. It does nothing else. After clearing the PC the microcode runs into the fetch routine.

- **0x003 – fetch**
  The fetch routine places the PC in the address output lines while postincrementing it, and then enables a memory read cycle. In doing so it relies on T2 being 0x00 (necessary for the ADC to behave like an INC in the oversimplified ALU), which is always true by design. After the fetch is done, the #decode flag is raised, which instructs the micromachine to take the value in the DI signal (data input from external memory) and load it into the IR and the microcode address counter, while setting the high address bit to 1. At the resulting address there will be a JSR μI pointing to the microcode for the 8080 opcode in question (the microcode assembler will make sure of that). The #decode flag will also clear registers T1 and T2.

- **0x007 – halt**
  Whenever a HALT instruction is executed, the #halt flag is raised, which when used in the same μI as flag #end, makes the the micromachine jump to this address. The μI at this address does nothing but raise flags #halt and #end. The micromachine will keep jumping to this address until the halt state is left, something which can only happen by reset or by interrupt. The #halt flag, when raised, sets the halt output signal, which will be cleared when the CPU exits the halt state.

2.2 Conditional jumps

There is a conditional branch microinstruction: TJSR. This instruction tests certain condition and, if the condition is true, performs exactly as JSR. Otherwise, it ends the microcode execution exactly as if the flag #end had been raised. This microinstruction has been made for the conditional branches and returns of the 8080 CPU and is not flexible enough for any other use. The condition tested is encoded in the register IR, in the field ccc (bits 5..3), as encoded in the conditional instructions of the 8080 – you can look them up in any 8080 reference. Flags are updated in the 2nd stage, so a TJSR cannot test the flags modified by the previous μI. But it is not necessary; this instruction will always be used to test conditions set by previous 8080 instructions, separated at least by the opcode fetch μIs, and probably many more. Thus, the condition flags will always be valid upon testing.

2.3 Implicit operations

Most micromachine operations happen only when explicitly commanded. But some happen automatically and have to be taken into account when coding the microprogram:

1. Register IR is loaded automatically when the flag #decode is raised. The microcode program counter is loaded automatically with the same value as the IR, as has been explained above. From that point on, execution resumes normally: the jump table contains normal JSR microinstructions.
2. T1 is cleared to 0x00 at reset, when the flag #decode is active or when the flag #clrt1 is used.
3. T2 is cleared to 0x00 at reset, when the flag #decode is active or when the flag #end is used.
4. Microcode flow control:
   (a) When flag #end is raised, execution continues at μcode address 0x0003.
   (b) When both flags #halt and #end are raised, execution continues at μcode address 0x0007, unless there is an interrupt pending.
   (c) Otherwise, when flag #ret is raised, execution continues in the address following the last JSR executed. If such a return is tried before a JSR has executed since the last reset, the results are undefined – this should never happen with the microcode source as it is now.
   (d) If none of the above flags are used, the next μI is executed.

Notice that both T1 and T2 are cleared at the end of the opcode fetch, so they are guaranteed to be 0x00 at the beginning of the instruction microcode. And T2 is cleared too at the end of the instruction microcode, so it is guaranteed clear for its use in the opcode fetch microcode. T1 can be cleared if a microinstruction so requires. Refer to the section on microinstruction flags.

3 Microinstructions

The microcode for the CPU is a source text file encoded in a format described below. This 'microcode source' is assembled by the microcode assembler (described later) which then builds a microcode table in VHDL format. There's nothing stopping you from assembling the microcode by hand directly on the VHDL source, and in a machine this simple it might have been better.

3.1 Microcode source format

The microcode source format is more similar to some early assembly language that to other microcodes you may have seen. Each non-blank, non-comment line of code contains a single microinstruction in the format informally described below:

```
< microinstruction line > :=
    [< label >][1] |
    < operand stage control > ; < ALU stage control > [; [< flag list >]] |
    JSR < destination address >| TJSR < destination address >


    < label > := {':' immediately followed by a common identifier}
    < destination address > := {an identifier defined as a label anywhere in the file}
    < operand stage control > := < op_reg > = < op_src > | NOP
    < op_reg > := T1 | T2
    < op_src > := < register > | DI | < IR register >
    < IR register > := {s}|{d}|{p}0|{p}1[2]
    < register > := _a|_b|_c|_d|_e|_h|_l|_f|_a|_ph|_pl|_x|_y|_z|_w|_sh|_sl
    < ALU stage control > := < alu_dst > = < alu_op > | NOP
    < alu_dst > := < register > | DO
    < alu_op > := add| adc| sub| sbb| and| orl| not| xrl| rla| rra| rlca| rrca| aaa|
    t1| rst| daa| cpc| sec| psw
    < flag list > := < flag > [, < flag > ...]
    < flag > := #decode|#di|#ei|#io|#auxcy|#clrt1|#halt|#end|#ret|#rd|#wr|#setacy
          #ld_al|#ld_addr|#fp_c|#fp_r|#fp_rc|#clr_acy [3]
```

Please bear in mind that this is just an informal description; I made it up from my personal notes and the assembler source. The ultimate reference is the microcode source itself and the assembler source.
Due to the way that flags have been encoded (there's less than one bit per flag in the microinstruction), there are restrictions on what flags can be used together. See section 3.4.

The assembler will complain if the source does not comply with the expected format; but syntax check is somewhat weak. In the microcode source you will see words like __reset, __fetch, etc. which don't fit the above syntax. Those were supposed to be assembler pragmas, which the assembler would use to enforce the alignment of the microinstructions to certain addresses. I finally decided not to use them and align the instructions myself. The assembler ignores them but I kept them as a reminder.

The 1st part of the μI controls the ALU operand stage; we can load either T1 or T2 with either the contents of the input signal DI, or the selected register from the register bank. Or, we can do nothing (NOP).
The 2nd part of the μI controls the ALU stage; we can instruct the ALU to perform some operation on the operands T1 and T2 loaded by this same instruction, in the previous stage; and we can select where to load the ALU result, eiher in the output register DO or in the register bank. Or we can do nothing of the above (NOP).

[1] Labels appear alone by themselves in a line
[2] Registers are specified by IR field
[3] There are some restrictions on the flags that can be used together

The write address for the register bank used in the 2nd stage has to be the same as the read address used in the 1st stage; that is, if both μI parts use the RB, both have to use the same address (the assembler will enforce this restriction). This is due to an early, silly mistake that I chose not to fix: there is a single μI field that holds both addresses.
This is a very annoying limitation that unduly complicates the microcode and wastes many microcode slots for no saving in hardware; I just did not want to make any major refactors until the project is working. As you can see in the VHDL source, the machine is prepared to use 2 independent address fields with little modification. I may do this improvement and others in a later version, but only when I deem the design 'finished' (since the design as it is already exceeds my modest performance target).

3.2 Microcode ALU operations

ALU operations			
Operation	encoding	result	notes
ADD	001100	T2 + T1	
ADC	001101	T2 + T1 + CY	
SUB	001110	T2 - T1	
SBB	001111	T2 T1 - CY	
AND	000100	T1 AND T2	
ORL	000101	T1 OR T2	
NOT	000110	NOT T1	
XRL	000111	T1 XOR T2	
RLA	000000	8080 RLC	
RRA	000001	8080 RRC	
RLCA	000010	8080 RAL	
RRCA	000011	8080 RAR	
T1	010111	T1	
RST	011111	8*IR(5..3)	as per RST instruction
DAA	101000	DAA T1	but only after executing 2 in a row
CPC	101100	UNDEFINED	CY complemented
SEC	101101	UNDEFINED	CY set
PSW	110000	PSW	

Notice that ALU operation DAA takes two cycles to complete; it uses a dedicated circuit with an extra pipeline stage. So it has to be executed twice in a row before taking the result – refer to microcode source for an example.
The PSW register is updated with the ALU result at every cycle, whatever ALU operation is executed - though every ALU operation computes flags by different means, as it is apparent in the case of CY. Which flags are updated, and which keep their previous values, is defined by a microinstruction field named flag_pattern. See the VHDL code for details.

3.3 Microcode binary format

Microcode word bitfields		
POS	VHDL NAME	PURPOSE
31..29	uc_flags1	Encoded flag of group 1 (see section on flags)
28..26	uc_flags2	Encoded flag of group 2 (see section on flags)
25	load_addr	Address register load enable (note 1)
24	load_al	AL load enable (note 1)
23	load_t1	T1 load enable
22	load_t2	T2 load enable
21	mux_in	T1/T2 source mux control (0 for DI, 1 for reg bank)
20..19	rb_addr_sel	Register bank address source control (note 2)
18..15	ra_field	Register bank address (used both for write and read)
14	clr_acy	Clear CY and AC – see explaination below (pipelined signal)
13..10	(unused)	Reserved for write register bank address, unused yet
11..10	uc_jmp_addr(7..6)	JSR/TJSR jump address, higher 2 bits
9..8	flag_pattern	PSW flag update control (note 3) (pipelined signal)
7	load_do	DO load enable (note 4) (pipelined signal)
6	we_rb	Register bank write enable (pipelined signal)
5..0	uc_jmp_addr(5..0)	JSR/TJSR jump address, lower 6 bits
5..0	(several)	Encoded ALU operation

- **Note 1: load_al**
  AL is a temporary register for the lower byte of the external 16 bit address. The memory interface (and the IO interface) assumes external synchronous memory, so the 16 bit address has to be externally loaded as commanded by load_addr. Note that both halves of the address signal load directly from the register bank output; you can load AL with PC, for instance, in the same cycle in which you modify the PC - AL will load with the pre-modified value.

- **Note 2 : rb_addr_sel**
  A microinstruction can access any register as specified by ra_field, or the register fields in the 8080 instruction opcode: S, D and RP (the microinstruction can select which register of the pair). In the microcode source this is encoded like this:

  {s} ⇒ 0 & SSS

  {d} ⇒ 0 & DDD

  {p}0 ⇒ 1 & PP & 0 (HIGH byte of register pair)

  {p}1 ⇒ 1 & PP & 1 (LOW byte of register pair)

  SSS = IR(5 downto 3) (source register)
  DDD = IR(2 downto 0) (destination register)
  PP = IR(5 downto 4) (register pair)

- **Note 3 : flag_pattern**
  Selects which flags of the PSW, if any, will be updated by the microinstruction:

  - When flag_pattern(0)='1', CY is updated in the PSW.
  - When flag_pattern(1)='1', all flags other than CY are updated in the PSW.

- **Note 4 : load_do**
  DO is the data ouput register that is loaded with the ALU output, so the load enable signal is pipelined.

- **Note 5 : JSR-H and JSR-L**
  These fields overlap existing fields which are unused in JSR/TJSR instructions (fields which can be used with no secondary effects).

3.4 Microcode flags

Flags is what I have called those signals of the microinstruction that you assert individually in the microcode source. Due to the way they have been encoded, I have separated them in two groups. Only one flag in each group can be used in any instruction. These are all the flags in the format thay appear in the microcode source:

- Flags from group 1: use only one of these
  - #decode : Load address counter and IR with contents of data input lines, thus starting opcode decoging.
  - #ei : Set interrupt enable register.
  - #di : Reset interrupt enable register.
  - #io : Activate io signal for 1st cycle.
  - #auxcy : Use aux carry instead of regular carry for this μI.
  - #clrt1 : Clear T1 at the end of 1st cycle.
  - #halt : Jump to microcode address 0x07 without saving return value, when used with flag #end, and only if there is no interrupt pending. Ignored otherwise.
- Flags from group 2: use only one of these
  - #setacy : Set aux carry at the start of 1st cycle (used for ++).
  - #end : Jump to microinstruction address 3 after the present m.i.
  - #ret : Jump to address saved by the last JST or TJSR m.i.
  - #rd : Activate rd signal for the 2nd cycle.
  - #wr : Activate wr signal for the 2nd cycle.
- Independent flags: no restrictions
  - #ld_al : Load AL register with register bank output as read by opn. 1 (used in memory and io access).
  - #ld_addr : Load address register (H byte = register bank output as read by operation 1, L byte = AL). Activate vma signal for 1st cycle.

  - #clr_acy : Clear PSW flags AC and CY, except for AND instructions (ALU operation = 000100), where AC is set. Meant to be used with flag #fp_rc for the logic instructions (AND, OR, XOR). See 5.4 for a note about compatibility to the original 8080.

- PSW update flags: use only one of these
  - #fp_r : This instruction updates all PSW flags except for C.
  - #fp_c : This instruction updates only the C flag in the PSW.
  - #fp_rc : This instruction updates all the flags in the PSW.

4 Notes on the microcode assembler

The microcode assembler is a Perl script (`util/uasm.pl`). Please refer to the comments in the script for a reference on the usage of the assembler.
I will admit up front that the microcode source format and the assembler program itself are a mess. They were hacked quickly and then often retouched but never redesigned, in order to avoid the 'never ending project' syndrome.
Please note that use of the assembler, and the microcode assembly source, is optional and perhaps overkill for this simple core. All you need to build the core is the vhdl source file.

The perl assembler itself accounted for more than half of all the bugs I caught during development. Though the assembler certainly saved me a lot of mistakes in the hand-assembly of the microcode, a half-cooked assembler like this one may do more harm than good. I expect that the program now behaves correctly; I have done a lot of modifications to the microcode source for testing purposes and I have not found any more bugs in the assembler. But you have been warned: don't trust the assembler too much (in case someone actually wants to mess with these things at all).
The assembler is a Perl program (`util/uasm.pl`) that will read a microcode text source file and write to stdout a microcode table in the form of a chunk of VHDL code. You are supposed to capture that output and paste it into the VHDL source (Actually, I used another perl script to do that, but I don't want to overcomplicate an already messy documentation).
The assembler can do some other simple operations on the source, for debug purposes. The invocation options are documented in the program file.
You don't need any extra Perl modules or libraries, any distribution of Perl 5 will do - earlier versions should too but might not, I haven't tested.

5 CPU details

5.1 Synchronous memory and i/o interface

The core is designed to connect to external synchronous memory similar to the internal fpga ram blocks found in the Spartan series. It can be used with asynchronous ram provided that you add

the necessary registers (I have used it with external SRAM included on a development board with no trouble).

Signal 'vma' is the master read/write enable. It is designed to be used as a synchronous rd/wr enable. All other memory/io signals are only valid when vma is active. Read data is sampled in the positive clock edge following deassertion of vma. Than is, the core expects external memory and io to behave as an internal fpga block ram would.
I think the interface is simple enough to be fully described by the comments in the header of the VHDL source file.

5.2 Interrupt response

Interrupt response has been greatly simplified, but it follows the outline of the original procedure. The biggest difference is that inta is active for the entire duration of the instruction, and not only the opcode fetch cycle.

Whenever a high value is sampled in line intr in any positive clock edge, an interrupt pending flag is internally raised. After the current instruction finishes execution, the interrupt pending flag is sampled. If active, it is cleared, interrupts are disabled and the processor enters an inta cycle. If inactive, the processor enters a fetch cycle as usual. The inta cycle is identical to a fetch cycle, with the exception that inta signal is asserted high.

The processor will fetch an opcode during the first inta cycle and will execute it normally, except the PC increment will not happen and inta will be high for the duration of the instruction. Note that though pc increment is inhibited while inta is high, pc can be explicitly changed (rst, jmp, etc.). After the special inta instruction execution is done, the processor resumes normal execution, with interrupts disabled.
The above means that any instruction (even XTHL, which the original 8080 forbids) can be used as an interrupt vector and will be executed normally. The core has been tested with rst, lxi and inr, for example.

Since there's no M1 signal available, feeding multi-byte instructions as interrupt vectors can be a little complicated. It is up to you to deal with this situation (i.e. use only single-byte vectors or make up some sort of cycle counter).

5.3 Instruction timing

This core is slower than the original in terms of clocks per instruction. Since the original 8080 was itself one of the slowest micros ever, this does not say much for the core. Yet, one of these clocked at 50MHz would outperform an original 8080 at 25 Mhz, which is fast enough for many control applications — except that there are possibly better alternatives.
A comparative table follows.

Instruction timing (core vs. original)						
Opcode	Intel 8080	Light8080		Opcode	Intel 8080	Light8080
MOV r1, r2	5	6		XRA M	7	9
MOV r, M	7	9		XRI data	7	9
MOV M, r	7	9		ORA r	4	6
MVI r, data	7	9		ORA M	7	9
MVI M, data	10	12		ORI data	7	9
LXI rp, data16	10	14		CMP r	4	6
LDA addr	13	16		CMP M	7	9
STA addr	13	16		CPI data	7	9
LHLD addr	16	19		RLC	4	5
SHLD addr	16	19		RRC	4	5
LDAX rp	7	9		RAL	4	5
STAX rp	7	9		RAR	4	5
XCHG	4	16		CMA	4	5
ADD r	4	6		CMC	4	5
ADD M	7	9		STC	4	5
ADI data	7	9		JMP	10	15
ADC r	4	6		Jcc	10	12/16
ADC M	7	9		CALL	17	29
ACI data	7	9		Ccc	11/17	12/30
SUB r	4	6		RET	10	14
SUB M	7	9		Rcc	5/11	5/15
SUI data	7	9		RST n	11	20
SBB r	4	6		PCHL	5	8
SBB M	7	9		PUSH rp	11	19
SBI data	7	9		PUSH PSW	11	19
INR r	5	6		POP rp	10	14
INR M	10	13		POP PSW	10	14
INX rp	5	6		XTHL	18	32
DCR r	5	6		SPHL	5	8
DCR M	10	14		EI	4	5
DCX rp	5	6		DI	4	5
DAD rp	10	8		IN port	10	14
DAA	4	6		OUT port	10	14
ANA r	4	6		HLT	7	5
ANA M	7	9		NOP	4	5
ANI data	7	9				
XRA r	4	6				

5.4 Binary compatibility to original 8080

Flag AC (auxiliary carry) does not work exactly as in the original 8080. In the original 8080, ANI and ANA don't clear AC but set it to the OR'ing of bits 3 of the ALU operands.

In this core, these two instructions instead set the AC flag to 1. In this, the core is compatible to the 8085 ad not to the 8080.

That is the only difference to the original 8080 that I am aware of. Unfortunately, the only test bench that I have available right now is not exhaustive enough to pick that kind of detail. Until I develop a stronger test bench, full compatibility to the 8080 can't be guaranteed.