Il Processore Mic-1

Dispensa didattica del corso di
Architettura dei Sistemi Digitali

Prof. Nicola Mazzocca
Ing. Alberto Moriconi

1 Il Livello Microarchitetturale

Il processore Mic-1 è un utile esempio didattico, presentato in [1] per due scopi principali:

1. mostrare come, usando elementi logici di base come quelli già studiati nel corso, sia possibile realizzare una microarchitettura che implementi un semplice set di istruzioni;

2. mostrare come anche la realizzazione di un sistema apparentemente complesso, come un processore, si riduca in realtà alla progettazione di un'unità operativa e di un'unità di controllo, e del modo in cui devono comunicare.

Il set di istruzioni implementato dal Mic-1 è un sottoinsieme di quello della Java Virtual Machine, denominato *IJVM* in quanto opera unicamente sugli interi.

Una particolarità di questo processore è quella di non disporre di registri generali; la sua architettura è infatti di tipo *a stack*: le sue istruzioni aritmetiche e logiche non hanno operandi espliciti, ma li prelevano da una struttura *last-in-first-out* allocata nella memoria principale, in cui devono essere posti in precedenza.

Per iniziare a comprendere come opera un processore di questo tipo, consideriamo ad esempio l'esecuzione dell'operazione di somma e assegnazione

```
a1 = a2 + a3;
```

che è tradotta in linguaggio assembly IJVM come

```
ILOAD a2
ILOAD a3
IADD
ISTORE a1
```

Lo stato dello stack durante l'esecuzione evolve come in fig. 1.1; inizialmente, le locazioni di memoria `a2` e `a3` contengono gli operandi, mentre la locazione `a1` è non inizializzata; dunque:

(a) la prima istruzione `ILOAD` legge il contenuto della locazione di memoria `a2` e ne esegue il push su stack;

(b) la seconda istruzione `ILOAD` legge il contenuto della locazione di memoria `a3` e ne esegue il push su stack;

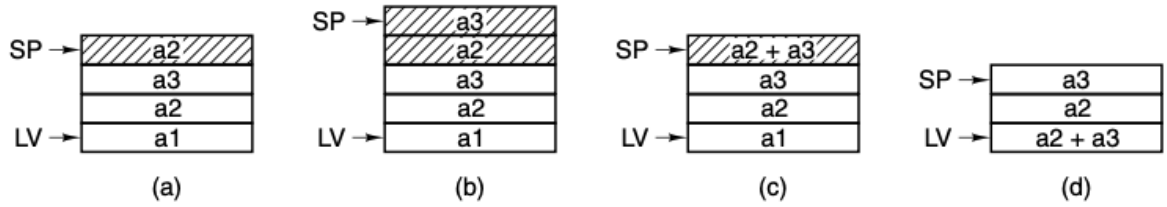

Figura 1.1: Evoluzione dello stack durante le operazioni di somma e assegnazione

(c) l'istruzione `IADD` esegue il pop dei due elementi in cima allo stack e il push della loro somma;

(d) l'istruzione `ISTORE` esegue il pop della somma e la scrive in memoria alla locazione `a1`.

Supponiamo che il processore si trovi al passo (c) e debba eseguire l'istruzione `IADD`; per farlo deve, in prima analisi:

1. eseguire un primo accesso in memoria, per prelevare il primo operando;
2. eseguire un secondo accesso in memoria, per prelevare il secondo operando;
3. sommare i due operandi;
4. eseguire un terzo accesso in memoria, per scrivere il risultato.

Per fare questo è necessario controllare gli accessi in memoria, l'ALU, ed eseguire delle operazioni di *book-keeping* (e.g. aggiornare il puntatore alla testa dello stack, che è modificato dalle operazioni di push e pop).

L'implementazione del processore Mic-1 che presentiamo è detta in *logica microprogrammata*: ciascuna istruzione IJVM è implementata come una sequenza di *microistruzioni* (detta talvolta *microprocedura*); tali sequenze compongono il *microprogramma*.

Nota 1.1

Il microprogramma è tipicamente memorizzato in una ROM interna al processore.

Per capire come è strutturata una microistruzione e come è possibile realizzare un microprograma che implementi un set di istruzioni, è necessario introdurre anzitutto l'unità operativa del processore.

1.1 L'Unità Operativa

L'unità operativa del processore che intendiamo progettare comprende l'ALU, i suoi ingressi, e le sue uscite (tra cui i registri che si interfacciano con la memoria), ed è mostrata in fig. 1.2.

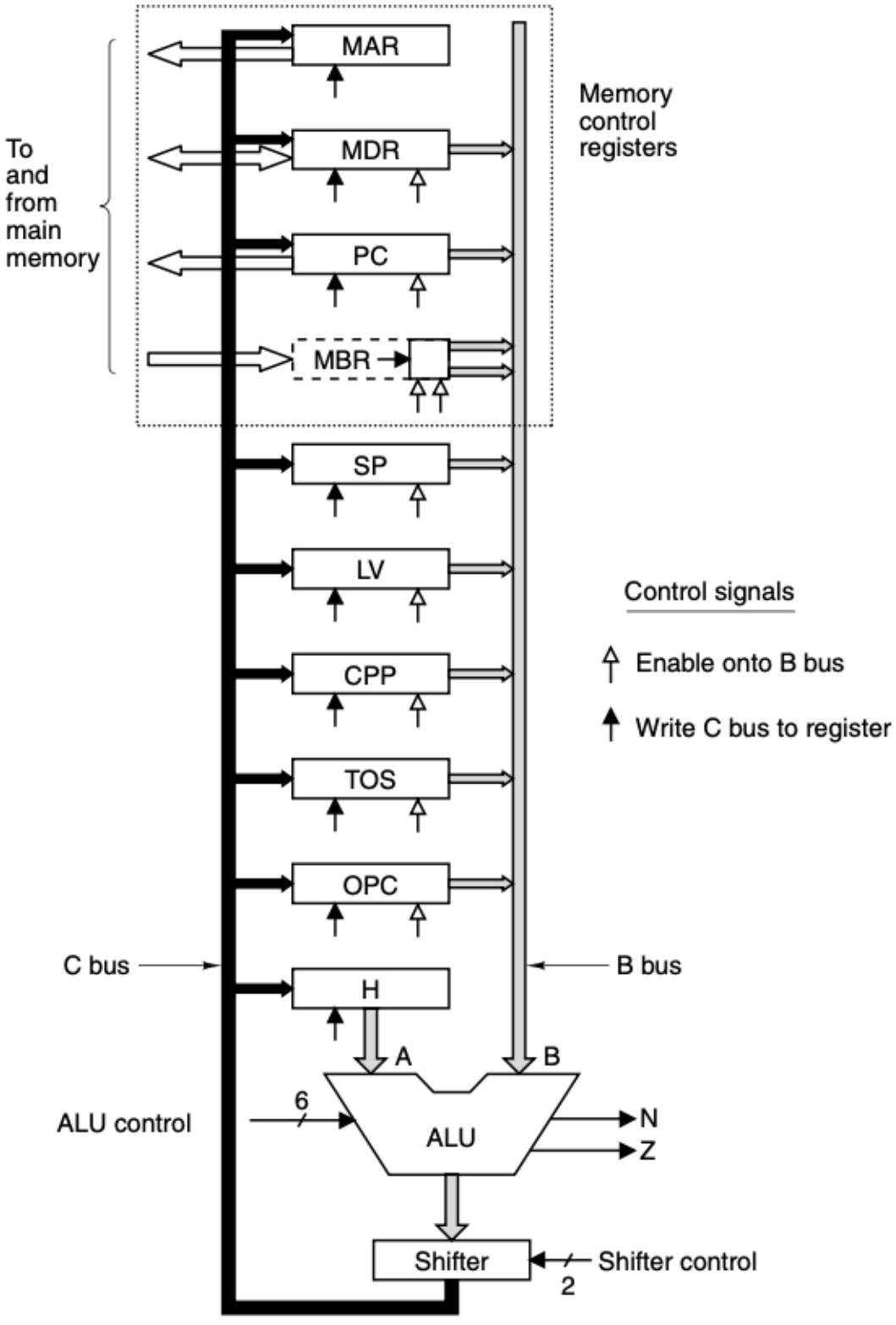

Figura 1.2: Parte operativa del processore Mic-1

I registri hanno dimensione di 32 bit e non sono accessibili al programmatore, ma solo al microprogramma.

Nota 1.2

Quando diciamo che i registri non sono accessibili al programmatore intendiamo che essi non fanno parte del *modello di programmazione*, i.e. non vengono utilizzati esplicitamente come operandi delle istruzioni. Vengono invece utilizzati dal microprogramma per implementare le istruzioni stesse.

L'unità operativa dispone di due bus, indicati con `B` e `C`, collegati rispettivamente al secondo ingresso e all'uscita dell'ALU; il primo ingresso dell'ALU è invece collegato esclusivamente al registro `H` (*holding*).

Con alcune eccezioni (i cui motivi diverranno chiari a breve), i registri dispongono di una coppia di segnali di controllo che permettono:

- di abilitare il collegamento del registro al bus `B`, rendendolo effettivamente l'operando `B` dell'ALU;
- di abilitare la scrittura sul registro del risultato fornito dall'ALU sul bus `C`.

Solo un registro può essere collegato al bus `B` in un determinato istante, mentre il risultato dell'ALU (i.e. il dato sul bus `C`) può essere scritto su più registri se necessario.

Nota 1.3

Il collegamento dei registri al bus `B` può essere implementato usando porte tri-state o un multiplexer; nel nostro caso, useremo un multiplexer in quanto gli FPGA di cui facciamo uso dispongono di buffer tri-state esclusivamente ai pad di IO.

Alcuni registri sono cablati in modo da poter essere usati solo per uno scopo specifico:

- i registri dell'interfaccia con la memoria:
  - `MAR` - memory address register;
  - `MDR` - memory data register;
  - `PC` - program counter;
  - `MBR` - memory byte register;
- il registro che mantiene il primo operando dell'ALU:
  - `H` - holding.

Gli altri registri sono funzionalmente equivalenti, ed i loro nomi sono assegnati sulla base dell'uso che se ne fa nel microprogramma:

- `SP` - stack pointer;
- `LV` - local variables;
- `CPP` - constant pool pointer;
- `TOS` - top of stack;
- `OPC` - scratch register.

1.1.1 L'Unità Aritmetico Logica

L'ALU del processore Mic-1 ha due ingressi, che denominiamo `A` e `B`; l'ingresso `A` è collegato esclusivamente al registro `H`, mentre l'ingresso `B` è collegato al bus omonimo, che può essere guidato dalle nove sorgenti indicate con una freccia grigia in fig. 1.2; le sue funzioni sono controllate mediante sei linee di controllo, che consideriamo asserite se poste a 1:

- `F0` e `F1` determinano l'operazione da eseguire;
- `ENA` e `ENB` abilitano singolarmente gli ingressi;
- `INVA` inverte l'ingresso `A`;
- `INC` incrementa di 1 il risultato.

Non tutte le combinazioni di questi segnali sono rilevanti; alcune delle più importanti per implementare il set di istruzioni IJVM sono riportate in tab. 1.3.

A questi segnali di controllo se ne aggiungono due ulteriori, utilizzati per eseguire lo shift del risultato dell'ALU prima che esso venga posto sul bus `C`:

- `SLL8` (*Shift Left Logical 8 bit*) esegue lo shift a sinistra di 8 bit, ponendo i bit meno significativi a `0`;
- `SRA1` (*Shift Right Arithmetic 1 bit*) esegue lo shift a destra di 1 bit, con estensione del segno (i.e. il bit più significativo non cambia).

Oltre a quella relativa al risultato, l'ALU dispone di due uscite *flag* ulteriori:

- il flag `N` è posto a 1 se il risultato è negativo;
- il flag `Z` è posto a 1 se il risultato è 0.

1.1.2 L'Interfaccia con la Memoria

Il processore Mic-1 può comunicare con la memoria mediante due interfacce:

- `MAR` e `MDR` controllano un'interfaccia word-addressable a 32 bit in lettura e scrittura;
- `PC` e `MBR` controllano un'interfaccia byte-addressable a 8 bit in sola lettura.

F_0	F_1	ENA	ENB	INVA	INC	Function
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	$\bar{A}$
1	0	1	1	0	0	$\bar{B}$
1	1	1	1	0	0	A + B
1	1	1	1	0	1	A + B + 1
1	1	1	0	0	1	A + 1
1	1	0	1	0	1	B + 1
1	1	1	1	1	1	B – A
1	1	0	1	1	0	B – 1
1	1	1	0	1	1	–A
0	0	1	1	0	0	A AND B
0	1	1	1	0	0	A OR B
0	1	0	0	0	0	0
1	1	0	0	0	1	1
1	1	0	0	1	0	–1

Figura 1.3: Segnali di controllo ALU e relative funzioni

Nota 1.4

Come vedremo in seguito, la prima interfaccia è utilizzata per accedere ai dati su cui operano le istruzioni IJVM, rappresentati su 32 bit; la seconda, invece, per prelevare le istruzioni IJVM dalla memoria, che possono avere una lunghezza variabile in multipli di byte (e richiedono dunque più accessi successivi in memoria). Gli indirizzi sono comunque rappresentati su 32 bit per entrambe le interfacce.

Le due interfacce funzionano in modo simile in lettura: l'indirizzo da cui si vuole leggere deve essere posto nel registro `MAR` (`PC`); al successivo fronte di salita del clock, il dato è disponibile nel registro `MDR` (`MBR`). Indirizzano però in modo diverso la memoria: scrivendo `0x00000002` nel registro `PC` e abilitando una lettura, il byte all'indirizzo `0x00000002` (i.e. il byte di indirizzo 2) viene letto e posto negli 8 bit meno significativi del registro `MBR`; scrivendo lo stesso indirizzo nel registro `MAR` e abilitando una lettura, i 4 byte `0x8-0x11` (i.e. la word di indirizzo 2) vengono letti e posti nel registro `MDR`.

L'interfaccia `MAR`/`MDR` dispone inoltre di un segnale di *write enable* (`WE`) che, se asserito al fronte di salita di clock, indica alla memoria di scrivere il dato contenuto in `MDR` all'indirizzo contenuto in `MAR`.

Il protocollo per l'interfaccia a 32 bit è mostrato in fig. 1.4.

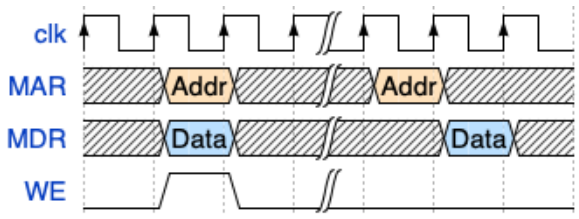

Figura 1.4: Protocollo di scrittura e lettura sull'interfaccia `MAR/MDR`

1.1.3 Implementazione VHDL dell'Unità Operativa

L'implementazione dell'unità operativa richiede di codificare in VHDL tre parti principali:

- i registri;
- i bus;
- l'ALU

Nota 1.5

Per il momento lasceremo in sospeso la struttura dell'entity relativa all'unità operativa, in quanto per capire quali segnali vengono esposti e perché è utile aver prima compreso anche il funzionamento dell'unità di controllo.

Implementazione dei Registri

I registri dell'unità operativa sono implementati mediante un singolo `process`, che ne gestisce la logica di reset e di scrittura mediante il bus `C`:

datapath.vhd

```
-- Registers
signal sp_reg   : reg_data_type;
signal lv_reg   : reg_data_type;
signal cpp_reg  : reg_data_type;

[...]

-- Processor registers
reg_proc : process(clk) is
begin
  if rising_edge(clk) then
```

```
    if reset = '1' then
      sp_reg  <= x"00000101";
      lv_reg  <= x"00000100";
      cpp_reg <= x"00000080";

      [...]

    else
      if c_to_reg_control(c_ctrl_mar) = '1' then
        mar_reg <= c_bus;
      end if;
      if c_to_reg_control(c_ctrl_pc) = '1' then
        pc_reg <= c_bus;
      end if;
      if c_to_reg_control(c_ctrl_sp) = '1' then
        sp_reg <= c_bus;
      end if;

      [...]

    end if;
  end if;
end process reg_proc;
```

Il tipo `reg_data_type` è introdotto come abbreviazione per un `std_logic_vector` di 32 bit (`31:0`); è definito, insieme ad altri tipi utili per l'implementazione del processore, nel file `common_defs.vhd`:

common_defs.vhd

```
-- Data widths
--! Processor register data width
constant reg_data_width      : positive := 32;

[...]

-- Subtypes
--! Processor register data
subtype reg_data_type is std_logic_vector(reg_data_width
 ↪  - 1 downto 0);
```

Il segnale `c_to_reg_control` è di tipo `c_ctrl_type`, anch'esso definito nel file `common_defs.vhd`, ed è indicizzato mediante una serie di costanti relative agli indici dei singoli segnali di attivazione:

common_defs.vhd

```
--! C bus control width
constant c_ctrl_width        : positive := 9;

[...]

--! C bus control type
subtype c_ctrl_type is std_logic_vector(c_ctrl_width - 1
 ↪  downto 0);

[...]

--! C control MAR bit
constant c_ctrl_mar          : natural := 0;
--! C control MDR bit
constant c_ctrl_mdr          : natural := 1;
--! C control PC bit
constant c_ctrl_pc           : natural := 2;
--! C control SP bit
constant c_ctrl_sp           : natural := 3;

[...]
```

Implementazione dell'ALU

In VHDL, l'`entity` declaration per l'ALU che abbiamo descritto può essere scritta come:

alu.vhd

```
entity alu is
  port (
    --! ALU control
    control       : in  alu_ctrl_type;
    --! ALU operand A
    operand_a     : in  reg_data_type;
    --! ALU operand B
```

```
    operand_b     : in  reg_data_type;
    --! ALU result
    sh_result     : out reg_data_type;
    --! Negative flag
    negative_flag : out std_logic;
    --! Zero flag
    zero_flag     : out std_logic
    );
end entity alu;
```

L'implementazione realizzata è puramente behavioral.

Implementazione dei Bus

In prima analisi, i bus sono implementati come dei `signal` VHDL:

datapath.vhd

```
-- Signals
signal a_bus : reg_data_type;
signal b_bus : reg_data_type;
signal c_bus : reg_data_type;
```

Punto critico nell'implementazione dei bus è però la disciplina di accesso utilizzata. Per quanto riguarda `A`, si tratta semplicemente dell'insieme di fili che collega il registro `H` al primo ingresso dell'ALU; il bus `C` invece è collegato all'uscita dello shifter della componente ALU:

datapath.vhd

```
-- ALU instantiation
alu : entity work.alu
  port map (
    control       => alu_control,
    operand_a     => a_bus,
    operand_b     => b_bus,
    sh_result     => c_bus,
    negative_flag => alu_n_flag,
    zero_flag     => alu_z_flag);
```

Come abbiamo visto nella sezione relativa ai registri, il bus `C` è connesso a tutti i registri e un segnale di abilitazione per ciascuno di essi funge da abilitazione in scrittura. Infine, il bus `B` è implementato mediante un multiplexer, implementato usando il costrutto VHDL `with/select` (*selected assignment*):

datapath.vhd

```
with reg_to_b_control select b_bus <=
  mdr_reg          when b_ctrl_mdr,
  pc_reg           when b_ctrl_pc,
  mbr_s            when b_ctrl_mbr,
  mbr_u            when b_ctrl_mbru,
  sp_reg           when b_ctrl_sp,
  lv_reg           when b_ctrl_lv,
  cpp_reg          when b_ctrl_cpp,
  tos_reg          when b_ctrl_tos,
  opc_reg          when b_ctrl_opc,
  (others => '0') when others;
```

1.1.4 Implementazione del Protocollo con la Memoria

La lettura e la generazione del segnale di *write enable* sono implementate nello stesso `process` utilizzato per gli altri registri; il dato proveniente dalla memoria è considerato come un'ulteriore possibile sorgente per il dato, in aggiunta al bus `C`:

datapath.vhd

```
-- Processor registers
reg_proc : process(clk) is
begin

  [...]

  if c_to_reg_control(c_ctrl_mdr) = '1' then
    mdr_reg <= c_bus;
  elsif rd_ff = '1' then
    mdr_reg <= mem_data_in;
  end if;

  [...]
```

```
  wr_ff <= mem_control(mem_ctrl_write);
end process;
```

I registri dell'interfaccia e il segnale di *write enable* sono dunque assegnati alle uscite dell'`entity`:

datapath.vhd

```
-- Output
mem_data_out  <= mdr_reg;
mem_data_addr <= mar_reg;
mem_data_we   <= wr_ff;
```

Nota 1.6

L'interfaccia istruzioni `PC/MBR` opera in modo simile in lettura, e non prevede operazioni di scrittura.

1.2 Le Microistruzioni

Per controllare l'unità operativa del processore, presentata in fig. 1.2, sono necessari 29 segnali:

- 9 segnali per controllare la scrittura dei dati dal bus `C` ai registri;
- 9 segnali per controllare quale registro è collegato al bus `B` e va in ingresso all'ALU;
- 8 segnali per controllare l'ALU;
- 2 segnali per abilitare lettura/scrittura sull'interfaccia `MAR/MDR`;
- 1 segnale per abilitare il fetch sull'interfaccia `PC/MBR`.

Poiché al più un registro alla volta può essere collegato al registro `B`, non c'è bisogno di utilizzare effettivamente 9 segnali: dato che al più uno di essi può essere asserito, le 10 configurazioni possibili possono essere codificate su $\lceil \log_2 10 \rceil = 4$ bit.

I 24 segnali individuati permettono di controllare l'unità operativa per un ciclo di clock; per completare una microistruzione, è necessario aggiungere due campi aggiuntivi, che chiamiamo `Addr` e `JAM`, che descriveremo approfonditamente quando parleremo dell'unità di controllo; per il momento, è sufficiente sapere che sono necessari a determinare la microistruzione da eseguire nel ciclo di clock successivo.

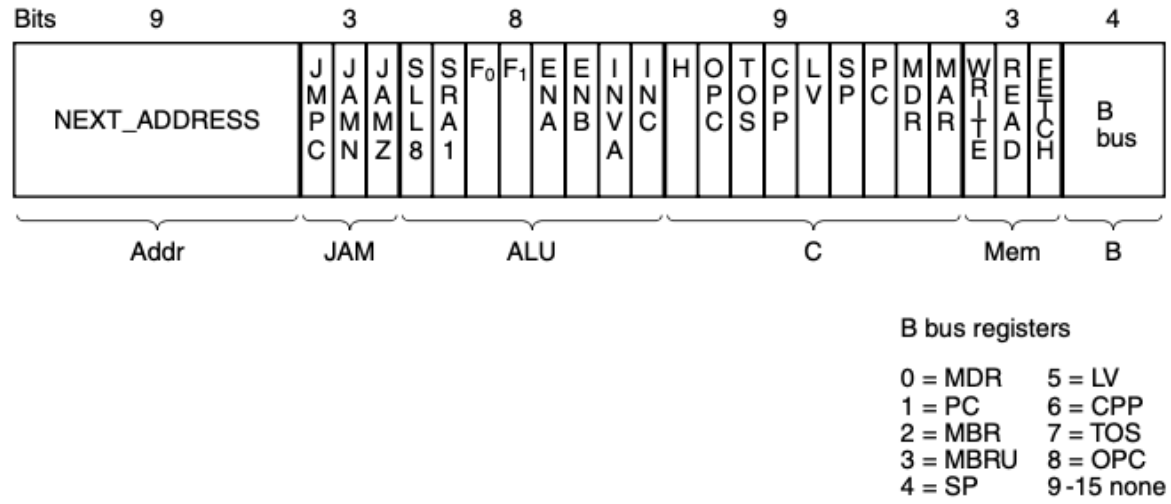

Figura 1.5: Il formato delle microistruzioni del Mic-1

In fig. 1.5 è presentato il formato che adotteremo per le microistruzioni, rappresentate su 36 bit; come anticipato, ogni microistruzione comprende i seguenti campi:

- `Addr` - Indirizzo di una potenziale prossima microistruzione;
- `JAM` - Determina come è selezionata la prossima microistruzione;
- `ALU` - Controllo dell'ALU e dello shifter;
- `C` - Controlla quali registri vengono scritti dal bus `C`;
- `Mem` - Controlla le operazioni di memoria;
- `B` - Seleziona il registro connesso al bus `B` (codifica in fig. 1.5).

1.2.1 Esempio pratico: l'istruzione `IADD`

Ora che abbiamo un'idea del funzionamento dell'unità operativa e della struttura di una microistruzione, possiamo analizzare più nel dettaglio il modo in cui viene eseguita un'istruzione IJVM. Consideriamo ad esempio l'istruzione `IADD`: quando l'abbiamo introdotta, abbiamo detto che "esegue il pop dei due elementi in cima allo stack e il push della loro somma"; questo significa che il processore, in prima analisi, dovrebbe eseguire due accessi in memoria in lettura, un'operazione di somma con l'ALU ed un accesso in memoria in scrittura. In pratica, però, l'implementazione del microprogramma Mic-1 mantiene due utili invarianti:

- il registro `SP` (*Stack Pointer*), al termine dell'esecuzione di un'istruzione IJVM, contiene sempre l'indirizzo dell'elemento in testa allo stack;

- il registro `TOS` (*Top of Stack*), al termine dell'esecuzione di un'istruzione IJVM, contiene sempre il valore dell'elemento in testa allo stack.

È dunque sufficiente un singolo accesso in memoria, e l'istruzione può essere eseguita in tre cicli di clock (i.e. tre microistruzioni); supponendo che lo stack si trovi nello stato in fig. 1.1 (b), per quanto detto in precedenza `TOS` contiene il valore `a3`; da questo stato:

1. il contenuto di `SP` viene decrementato di 1 e scritto sia in `SP` che in `MAR`, e si avvia l'operazione di lettura; si noti che `SP` punta ora alla posizione da cui deve essere letto il valore di `a2` e in cui in seguito sarà scritta la somma `a2 + a3`;
2. il contenuto di `TOS` è posto nel registro `H`;
3. i dati contenuti in `TOS` e `MDR` (dato proveniente dalla memoria) sono sommati e posti sia in `TOS` che in `MDR`, si avvia una operazione di scrittura.

È possibile verificare manualmente che queste operazioni rispettano le invarianti enunciate in precedenza, e che il valore in cima allo stack è ora proprio la somma dei due operandi che vi erano stati posti in precedenza.

Trascurando per ora i campi `Addr` e `JAM`, siamo in grado di codificare gli altri campi delle microistruzioni:

1.
   - `ALU`: `110110` (decremento dell'operando `B`);
   - `C`: `000001001` (bus `C` scrive su `SP` e `MAR`);
   - `Mem`: `010` (lettura dati);
   - `B`: `0100` (`SP` controlla il bus `B`).

   viene decrementato lo Stack Pointer, viene scritto in SP e MAR, e viene mandata la richiesta di leggere il dato dalla memoria. TUTTO IN UN SOLO CICLO DI CLOCK
2.
   - `ALU`: `010100` (l'operando `B` passa invariato);
   - `C`: `100000000` (bus `C` scrive su `H`);
   - `Mem`: `000` (nessuna operazione di memoria);
   - `B`: `0111` (`TOS` controlla il bus `B`).

   Il primo operando (che si trova in TOS) deve passare per il bus B, poi per l'ALU (dove non viene alterato) e poi scritto nel registro H (primo operando fisso dell'alu)
3.
   - `ALU`: `111100` (somma degli operandi `A` e `B`);
   - `C`: `001000010` (bus `C` scrive su `MDR` e `TOS`);
   - `Mem`: `100` (scrittura dati);
   - `B`: `0000` (`MDR` controlla il bus `B`).

   viene completata l'operazione di lettura dalla memoria (avviata nell'operazione 1), il dato sta in MDR. MDR scrive sul bus B, l'ALU effettua la somma tra A e B e il risultato viene posto in MDR e TOS. TUTTO IN UN CICLO DI CLOCK.

1.2.2 Il linguaggio MAL: parte 1

Scrivere a mano le microistruzioni è perfettamente possibile, ma è molto semplice commettere errori; inoltre, il microprogramma così ottenuto sarebbe tedioso da comprendere e modificare.

DEST = H
DEST = SOURCE
DEST = $\overline{\text{H}}$
DEST = $\overline{\text{SOURCE}}$
DEST = H + SOURCE
DEST = H + SOURCE + 1
DEST = H + 1
DEST = SOURCE + 1
DEST = SOURCE – H
DEST = SOURCE – 1
DEST = –H
DEST = H AND SOURCE
DEST = H OR SOURCE
DEST = 0
DEST = 1
DEST = –1

Figura 1.6: Le operazioni permesse nel linguaggio MAL

Il linguaggio usato in [1] per semplificare la scrittura del microprogramma è denominato MAL (*MicroAssembly Language*); un tool detto *microassemblatore* esegue dunque la traduzione nel formato di fig. 1.5, completamente equivalente.

Introduciamo anzitutto la notazione necessaria a specificare il controllo dei bus `B` e `C` e dell'ALU; le operazioni permesse sono riportate in fig. 1.6, dove `SOURCE` è uno dei registri connessi al bus `B` e `DEST` una combinazione di registri connessi al bus `C`, eventualmente separati dal simbolo =; ad esempio, per incrementare il contenuto del registro `SP` e scriverlo sia in `SP` che in `MAR`, si scrive:

```
SP = MAR = SP + 1
```

A ciascuna di queste operazioni può essere applicato uno shift a sinistra di 1 byte aggiungendo `<< 8`, ad esempio:

```
H = MBR << 8
```

Una seconda parte della microistruzione, separata dalla precedente da un punto e virgola (`;`), riguarda le istruzioni di memoria; lettura e scrittura sull'interfaccia dati sono indicate rispettivamente da `rd` o `wr`, mentre la lettura sull'interfaccia istruzioni è indicata da `fetch`; in una singola microistruzione è possibile avere un'operazione per ciascuna delle due interfacce, eventualmente separate da `;`.

A questo punto siamo in grado effettivamente di tradurre in linguaggio MAL le microistruzioni dell'istruzione `IADD`:

```
MAR = SP = SP - 1; rd
H = TOS
MDR = TOS = MDR + H; wr
```

Questo frammento di microprogramma è tuttavia incompleto; per capire la funzione dei campi `Addr` e `JAM` è necessario introdurre l'unità di controllo del Mic-1.

1.3 L'Unità di Controllo

In fig. 1.7 è riportato il diagramma completo del processore Mic-1, in cui oltre all'unità operativa sono riportate le componenti dell'unità di controllo.

L'unità di controllo del Mic-1 si comporta come un *sequencer*, producendo in ciascun ciclo:

1. lo stato dei segnali di controllo;
2. l'indirizzo della prossima microistruzione da eseguire.

Il microprogramma è effettivamente memorizzato in una memoria a sola lettura, interna al processore, detta *control store*; in pratica, il control store contiene le microistruzioni del microprogramma allo stesso modo in cui la memoria principale contiene le istruzioni (di livello ISA) del programma da eseguire.

Anche l'accesso al control store richiede un'interfaccia, controllata dai due registri `MPC` (*MicroProgram Counter*) e `MIR` (*MicroInstruction Register*); non è necessario un segnale di lettura, in quanto il control store è letto continuamente ad ogni ciclo.

Un'importante differenza è legata al fatto che, mentre le istruzioni del programma sono eseguite generalmente in ordine sequenziale (a meno ovviamente di istruzioni di salto), ciascuna microistruzione specifica esplicitamente l'indirizzo della successiva, riportandolo nel campo `Addr`.

Le microistruzioni forniscono tuttavia un supporto al controllo dei salti mediante il campo `JAM`; i due bit `JAMN` e `JAMZ` sono combinati con i valori dei flag `N` e `Z` dell'ALU, e permettono di modificare il bit più significativo dell'`MPC`:

```
MPC[8] = (JAMZ AND Z) OR (JAMN AND N) OR Addr[8]
```

Inoltre, se il bit `JMPC` è asserito, gli 8 bit meno significativi di `MPC` sono messi in `OR` bit-a-bit con gli 8 bit contenuti nel registro `MBR`.

1.3.1 Il linguaggio MAL: parte 2

Possiamo a questo punto introdurre le parti mancanti del linguaggio MAL; osserviamo il microcodice completo per l'istruzione `IADD`:

```
iadd = 0x65:
    MAR = SP = SP - 1; rd
    H = TOS
    MDR = TOS = MDR + H; wr; goto main
```

Abbiamo aggiunto una label, `iadd`, che permette di identificare la locazione del control store, seguita dall'indirizzo nel control store a partire dal quale verra posizionato

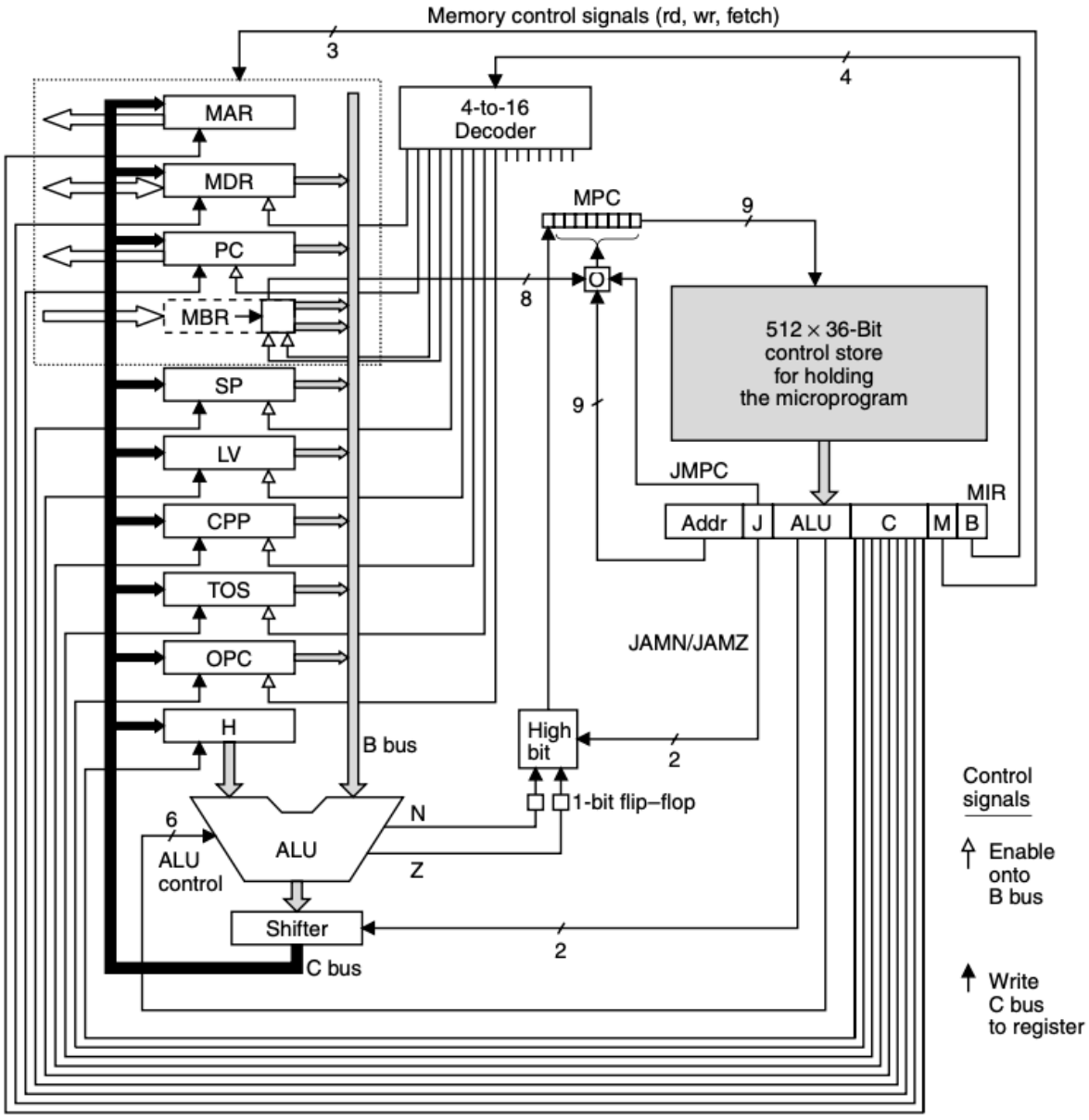

Figura 1.7: Diagramma completo del processore Mic-1

il microcodice seguente (dunque 0x65-0x67); in assenza di indicazioni esplicite, il campo `Addr` contiene semplicemente l'indirizzo della microistruzione successiva (dunque in questo caso, le prime due istruzioni, situate agli indirizzi 0x65 e 0x66 nel control store, avranno campo `Addr` posto rispettivamente a 0x66 e 0x67).

Consideriamo invece l'ultima microistruzione; abbiamo aggiunto un'ulteriore parte alla microistruzione, che influenza il suo campo `Addr`:

```
goto main
```

indica al programma microassemblatore, che traduce il linguaggio MAL nel formato delle microistruzioni, che il campo `Addr` per questa microistruzione deve essere posto uguale all'indirizzo della microistruzione che ha come label `main`.

Osserviamo anche il microcodice relativo alla label `main`:

```
main:
    PC = PC + 1; fetch; goto (MBR)
```

Questa microistruzione è particolarmente importante, in quanto ogni istruzione IJVM si conclude tornando ad essa; la sua funziona è incrementare il program counter, fare il fetch del successivo byte di istruzione e usare il contenuto del registro `MBR` come prossimo valore del microprogram counter; il campo

```
goto (MBR)
```

è codificato usando il bit `JMPC` nel campo `JAM`.

È chiaro da quanto detto che l'indirizzo che assegniamo ad una istruzione IJVM nel control store deve essere uguale alla sua codifica in linguaggio macchina prodotta dall'assemblatore IJVM: l'istruzione `IADD` è codificata come `0x65`, e memorizzata proprio all'indirizzo `0x65` nel control store; quando la microistruzione `main` esegue il fetch del byte `0x65`, l'`MPC` è settato a `0x65` e il microcodice relativo all'istruzione `IADD` viene eseguito.

Per concludere, il controllo di flusso basato sui flag dell'ALU è scritto in linguaggio MAL come:

```
if (flag) goto T; else goto F
```

dove `flag` può essere `N` o `Z`, mentre `T` e `F` sono due label.

1.3.2 Implementazione dell'Unità di Controllo

Il control store presenta un'interfaccia molto semplice:

control_store.vhd

```
entity control_store is
  port (
    --! Address of the desired word
    address : in  ctrl_str_addr_type;
    --! Content of the addressed word
```

```
    word     : out ctrl_str_word_type
    );
end entity control_store;
```

Le microistruzioni sono inserite direttamente all'interno del codice:

control_store.vhd

```
constant words : ctrl_str_type := (
--BEGIN_WORDS_ENTRY
0 => "00000011000000000000000000000001001",
1 => "01011110000000000000000000000001001",
2 => "00000000000000000000000000000000000",

[...]

--END_WORDS_ENTRY
);
```

Tutti i tipi sono definiti, come negli altri casi, nel file `common_defs.vhd`:

common_defs.vhd

```
--! Control store address
subtype ctrl_str_addr_type is
 ↪  std_logic_vector(ctrl_str_addr_width - 1 downto 0);
--! Control store word
subtype ctrl_str_word_type is
 ↪  std_logic_vector(ctrl_str_word_width - 1 downto 0);

[...]

--! Control store word
subtype ctrl_str_word_type is
 ↪  std_logic_vector(ctrl_str_word_width - 1 downto 0);

[...]

--! Control store content
type ctrl_str_type is array (ctrl_str_words - 1 downto 0)
 ↪  of ctrl_str_word_type;
```

L'implementazione della control unit è molto semplice: un `process` aggiorna il registro `MIR` con la microistruzione corrente, i cui bit del campo `Addr`, opportunamente mascherati, costituiscono il registro "virtuale" `MPC`:

control_unit.vhd

```
-- Registers
reg_proc : process(clk) is
begin
  if rising_edge(clk) then
    if reset = '1' then
      mir_reg <= "00000000100000000000000000000001001";
      n_ff    <= '0';
      z_ff    <= '0';
    else
      mir_reg <= control_store_word;
      n_ff    <= alu_n_flag;
      z_ff    <= alu_z_flag;
    end if;
  end if;
end process reg_proc;

-- MPC virtual register
ctrl_nxt_addr_no_msb <=
 ↪  mir_reg(ctrl_nxt_addr_no_msb_type'range);
jmpc_addr <= ctrl_nxt_addr_no_msb or mbr_reg_in when
 ↪  mir_reg(ctrl_jmpc) = '1' else ctrl_nxt_addr_no_msb;
high_bit <= (alu_n_flag and mir_reg(ctrl_jamn)) or
 ↪  (alu_z_flag and mir_reg(ctrl_jamz));
mpc_virtual_reg <= (mir_reg(ctrl_nxt_addr_msb) or
 ↪  high_bit) & jmpc_addr;
```

Il decoder per la selezione del registro che controlla il bus `B` è implementato come un `process` puramente combinatorio:

control_unit.vhd

```
-- B_BUS control decoder
reg_to_b_decoder : process(mir_reg(ctrl_b'range)) is
begin
  reg_to_b_decoder_out <= (others => '0');
```

```
    if unsigned(mir_reg(ctrl_b'range)) < b_ctrl_width then
      reg_to_b_decoder_out(to_integer(
        unsigned(mir_reg(ctrl_b'range)))) <= '1';
    end if;
  end process reg_to_b_decoder;
```

1.3.3 Collegarmento tra UO e UC

Siamo finalmente in grado di definire le `entity` relative all'unità operativa e all'unità di controllo.

Entrambe le unità devono ricevere:

- un segnale di clock;
- un segnale di reset.

Questi segnali sono comuni.

L'unità di controllo fornisce all'unità operativa i segnali relativi:

- al controllo dell'ALU;
- al controllo del bus `C`;
- al controllo del bus `B`;
- al controllo della memoria.

Riceve invece da essa:

- i flag dell'ALU;
- il contenuto del registro `MBR`.

control_unit.vhd

```
 entity control_unit is
  port (
    --! Clock
    clk              : in  std_logic;
    --! Synchronous active-high reset
    reset            : in  std_logic;
    --! Content of the MBR register
    mbr_reg_in       : in  mbr_data_type;
    --! ALU negative flag
```

```
    alu_n_flag       : in  std_logic;
    --! ALU zero flag
    alu_z_flag       : in  std_logic;
    --! Control signals for the ALU
    alu_control      : out alu_ctrl_type;
    --! Control signals for the C bus
    c_to_reg_control : out c_ctrl_type;
    --! Control signals for memory operations
    mem_control      : out mem_ctrl_type;
    --! Control signals for the B bus
    reg_to_b_control : out b_ctrl_type
    );
end entity control_unit;
```

In aggiunta a questi segnali, ci aspettiamo di trovare nell'`entity` dell'unità operativa anche i porti relativi all'interfaccia con la memoria:

- indirizzo dati;
- ingresso dati;
- uscita dati;
- write enable dati;
- indirizzo istruzioni;
- ingresso istruzioni.

datapath.vhd

```
entity datapath is
  port (
    --! Clock
    clk              : in  std_logic;
    --! Synchronous active-high reset
    reset            : in  std_logic;
    --! Control signals for the ALU
    alu_control      : in  alu_ctrl_type;
    --! Control signals for the C bus
    c_to_reg_control : in  c_ctrl_type;
    --! Control signals for memory operations
    mem_control      : in  mem_ctrl_type;
```

```
    --! Control signals for the B bus
    reg_to_b_control : in  b_ctrl_type;
    --! Content of the MBR register
    mbr_reg_out      : out mbr_data_type;
    --! ALU negative flag
    alu_n_flag       : out std_logic;
    --! ALU zero flag
    alu_z_flag       : out std_logic;
    --! Memory data write enable
    mem_data_we      : out std_logic;
    --! Port for memory data read
    mem_data_in      : in  reg_data_type;
    --! Port for memory data write
    mem_data_out     : out reg_data_type;
    --! Memory address for memory data operations
    mem_data_addr    : out reg_data_type;
    --! Port for memory instruction read
    mem_instr_in     : in  mbr_data_type;
    --! Memory address for memory instruction fetch
    mem_instr_addr   : out reg_data_type
    );
end entity datapath;
```

Infine, introduciamo il componente `processor` all'interno del quale sono istanziati sia il `datapath` che la `control_unit`. Tutte le coppie di porti corrispondenti sono connesse internamente, mentre sono esposti quelli relativi:

- al clock;
- al reset;
- all'interfaccia con la memoria.

processor.vhd

```
entity processor is
  port (
    --! Clock
    clk             : in  std_logic;
    --! Synchronous active-high reset
    reset           : in  std_logic;
    --! Memory data write enable
    mem_data_we     : out std_logic;
```

```
    --! Port for memory data read
    mem_data_in    : in  reg_data_type;
    --! Port for memory data write
    mem_data_out   : out reg_data_type;
    --! Memory address for memory data operations
    mem_data_addr  : out reg_data_type;
    --! Port for memory instruction read
    mem_instr_in   : in  mbr_data_type;
    --! Memory address for memory instruction fetch
    mem_instr_addr : out reg_data_type
    );
end entity processor;
```

2 La Toolchain Mic-1

2.1 Installazione

La procedura seguente permette di installare il microassemblatore `mal` e l'assemblatore `ajvm`, oltre a una serie di dipendenze aggiuntive per poter simulare il processore. Possono esserci piccole variazioni a seconda del sistema operativo usato, quindi è consigliato utilizzare come versione di riferimento Ubuntu 20.10.

È necessario per prima cosa installare i tool essenziali per la compilazione, il version control system Git, il simulatore GHDL, il visualizzatore di forme d'onda gtkwave e il Java Development Kit.

```
$ sudo apt install gcc g++ make cmake git ghdl-gcc gtkwave wget
  ↪ apt-transport-https gnupg
$ wget -qO -
  ↪ https://adoptopenjdk.jfrog.io/adoptopenjdk/api/gpg/key/public |
  ↪ sudo apt-key add -
$ echo "deb https://adoptopenjdk.jfrog.io/adoptopenjdk/deb groovy
  ↪ main" | sudo tee /etc/apt/sources.list.d/adoptopenjdk.list
$ sudo apt update
$ sudo apt install adoptopenjdk-11-hotspot
```

Conviene dunque creare nella vostra home una nuova cartella per i repository da scaricare e una per i tool compilati:

```
$ mkdir tools
$ mkdir esercitazione_mic
$ cd esercitazione_mic
```

È possibile a questo punto compilare i tool (ignorando eventuali warning in fase di test):

```
$ git clone https://github.com/albmoriconi/mal.git
$ cd mal
$ ./gradlew build
$ cd ..
```

```
$  git clone https://github.com/albmoriconi/ajvm.git
$  cd ajvm
$  ./gradlew build
$  cd ..
$  tar -xvf mal/build/distributions/mal.tar -C ~/tools/
$  tar -xvf ajvm/build/distributions/ajvm.tar -C ~/tools/
```

Per poter usare i tool nella shell attiva, le loro directory vanno aggiunte al path di ricerca dei binari, ad esempio aggiungendolo in coda al file `.bashrc` con il seguente comando:

```
$  echo 'export PATH="$HOME/tools/ajvm/bin:$HOME/tools/mal/bin:$PATH"'
    ↪  >> ~/.bashrc
```

Nota 2.1

Chiudere e riaprire la finestra del terminale per assicurarsi che il file `.bashrc` venga riletto, dunque riposizionarsi nella directory `esercitazione_mic` prima di continuare.

Infine, è possibile eseguire il clone del repository contenente il codice VHDL del processore, e predisporne la directory per la build:

```
$  git clone https://github.com/albmoriconi/amic-0.git
$  cd amic-0
$  cmake -B build
```

2.2 Struttura del Progetto

Da qui in avanti, ogni volta che faremo riferimento a un percorso sarà rispetto alla directory in cui è stato scaricato il repository `amic-0`.

La directory principale del repository è organizzata come segue:

- `src` - codice sorgente
  - `main` - implementazione (subdirectory ordinate per linguaggio)
  - `test` - testbench (subdirectory ordinate per linguaggio)
- `util` - utility necessarie per il flow di sviluppo (subdirectory ordinate per linguaggio)

- `build` - contiene i file prodotti dal build system (questa directory deve essere creata manualmente)

Per il momento non ci interessano le altre directory e file presenti; è importante tuttavia notare che ogni directory che contiene codice (eventualmente nelle subdirectory) contiene anche un file `CMakeLists.txt` che serve al build system per tenere traccia dei sorgenti e aggiungerli ai target opportuni.

2.3 Flow di Sviluppo

Si può adesso lanciare usare il tool `cmake` con una serie di target utili per lanciare microassemblatore, assemblatore e alcuni test.

In particolare, è possibile editare il microprogramma in linguaggio MAL e il programma in linguaggio IJVM situati rispettivamente nelle directory `src/main/mal` e `src/main/ajvm` e dunque rigenerare control store e RAM lanciando, dalla directory `build`, i target:

```
$   cmake --build build --target create_control_store
$   cmake --build build --target create_ram
```

Il target `check` lancia invece i testbench contenuti nella directory `src/test/vhdl`:

```
$   cmake --build build --target check
```

Attenzione 2.1

Modificando il programma contenuto nella RAM, il testbench `processor_tb.vhd` potrebbe fallire in quanto verifica il risultato prodotto dal programma precaricato; due possibili soluzioni sono:

- modificare le condizioni alle righe 97-100 in modo da verificare le nuove condizioni sul risultato; oppure
- più semplicemente, si possono rimuovere le stesse righe; in questo caso, se si usa il simulatore GHDL, sarà necessario terminare manualmente l'esecuzione del tasto (`Ctrl + C`) o modificare gli argomenti a linea di comando per specificare la durata del test.

Un esempio dei risultati della suite di test è riportato in fig. 2.1. L'esecuzione dei test produce anche le waveform relative alle esecuzioni dei testbench, come quella ripor-

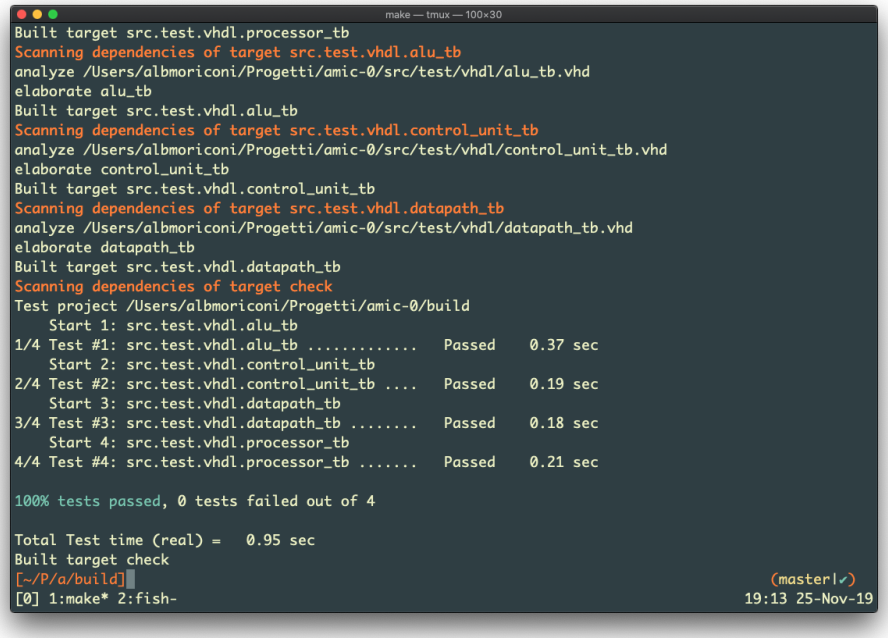

Figura 2.1: Risultati dei testbench

tata in fig. 2.2, che possono essere trovate nella directory `build/src/test/vhdl` e visualizzate utilizzando il tool `gtkwave`.

Attenzione 2.2

Il comando `make check` potrebbe non funzionare con tutte le versioni di GHDL e CMake. Qualora si riscontrassero errori nell'esecuzione dei test con questa modalità, è sufficiente importare manualmente i sorgenti `.vhd` contenuti nella directory `amic-0/src/main/vhdl` e il testbench `amic-0/src/test/vhdl/processor_tb.vhd` in un progetto ISE o Vivado, e utilizzare il simulatore Xilinx.

Per aggiungere altri testbench alla suite è sufficiente salvarli nella directory `test/vhdl` e modificare il file `test/vhdl/CmakeLists.txt` aggiungendo alla macro `add_vhdl_test_sources` un'entry relativa al nuovo testbench.

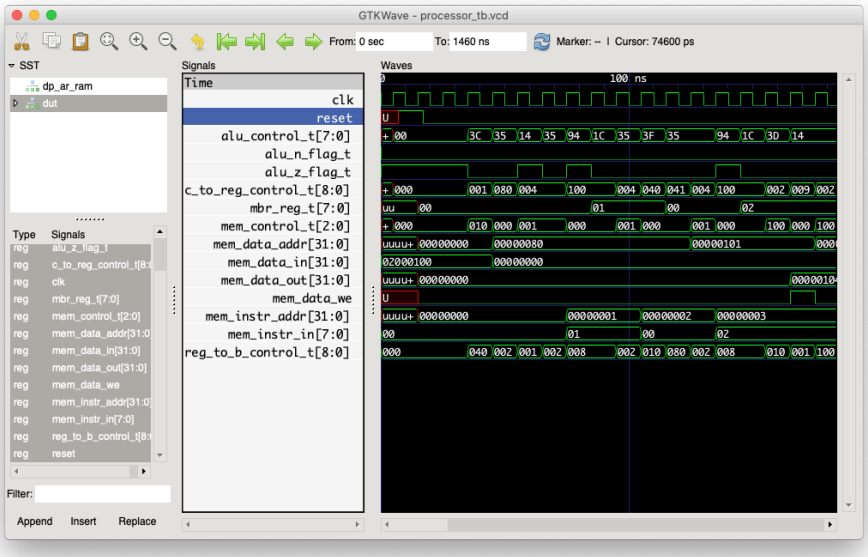

Figura 2.2: Waveform del testbench `processor_tb.vhd`

Bibliografia

[1] Andrew S. Tanenbaum and Todd Austin. *Structured Computer Organization*. Pearson, 2013.

[2] Gianni Conte, Antonino Mazzeo, Nicola Mazzocca and Paolo Prinetto. *Architettura dei Calcolatori*. CittàStudi, 2015.