CS201 W'00 Lecture Notes
Lecture 7

Mic-1 Block Diagram

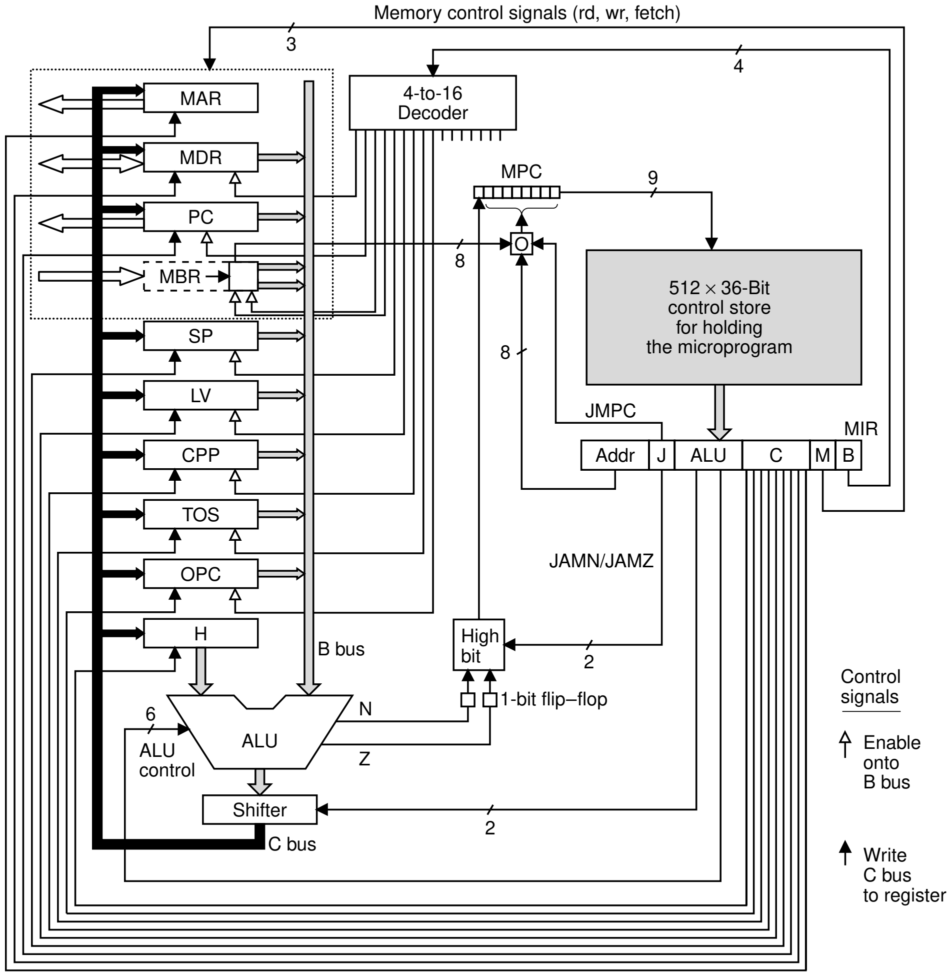


Figure 4-6. The complete block diagram of our example microarchitecture, the Mic-1.

Example Instruction

What steps are needed for `IADD`?

Assume that IADD opcode is already in MBR.

We must:

- Adjust SP to SP-1.
- Read contents of memory at (new) SP.
- Add those contents to TOS, replacing old TOS, and writing to memory at (new) SP.
- Increment PC and fetch opcode at new PC into MBR ready to start next instruction.

Recall that all memory reads and fetches require an extra cycle.

Can do this in four cycles.

Controlling One Cycle

For each cycle, must specify all control line settings.

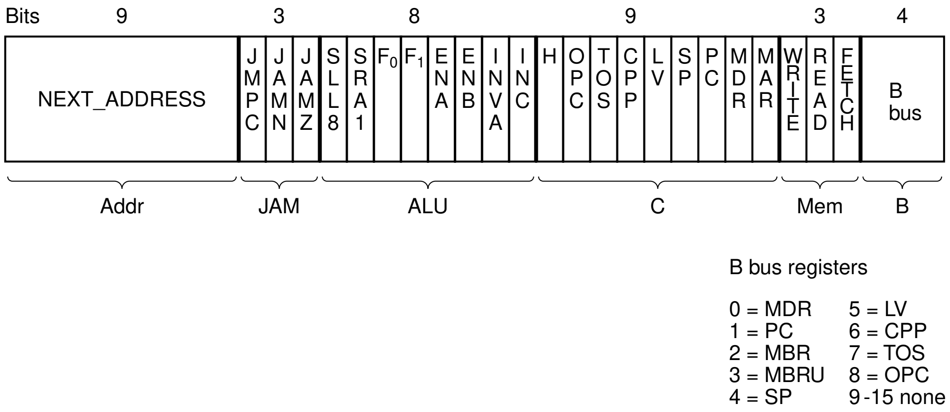


Figure 4-5. The microinstruction format for the Mic-1.

iadd1: Decrement SP and set MAR to read mem at new SP.

C.SP = 1
C.MAR = 1
C.x = 0 (all other x)
B = 4 (SP)
F_0 = 1
F_1 = 1
ENVA = 0
ENVB = 1
INVA = 1
INC = 0
SLL8 = 0
LRA1 = 0
FETCH = 0
RD = 1
WR = 0

Example (Continued)

To simplify notation: indicate only 1's (not 0's), give mnemonic for B, and give function summary for 6 ALU control lines.

iadd1 : Decrement SP and set MAR to read contents of new SP.

```
C.SP = 1
C.MAR = 1
B = SP
ALU = B - 1
RD = 1
```

iadd2: Move TOS to H.

```
C.H = 1
B = TOS
ALU = B
```

iadd3: MDR now contains contents of new SP; add to H and store in memory and back in TOS.

```
C.MDR = 1
C.TOS = 1
B = MDR
ALU = B + A
WR = 1
```

Example (continued)

Main1: Adjust PC and fetch next instruction.

```
C.PC = 1
B = PC
ALU = B + 1
FETCH = 1
```

In fact, we have to do Main1 for **every** opcode, and we need to do it earlier to guarantee that MBR has the new opcode in time to dispatch on it, so we'll normally do it **first**.

Simplifying Notation Again

This representation is good for keeping close track of control lines, but it is hard to grasp the purpose of each step.

Tanenbaum introduces an assembler-like notation to describe the net effect of each step as a (multi-)assignment, with optional memory control assertions.

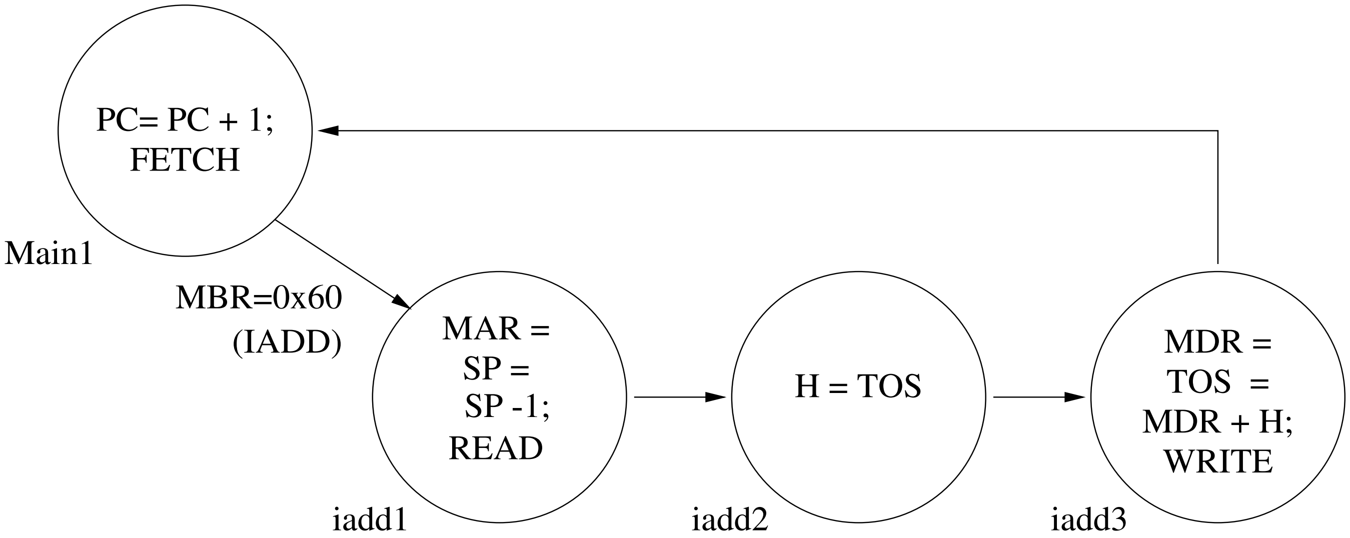
```
iadd1: MAR = SP = SP - 1; rd
iadd2: H = TOS
iadd3: MDR = TOS = MDR + H; wr
Main1: PC = PC + 1; fetch
```

This is compact and readable, **but** must be careful not to "invent" steps that are not actually supported by the hardware!

There's a bit more to the **mal** assembly language (more later)

Finite State Machines

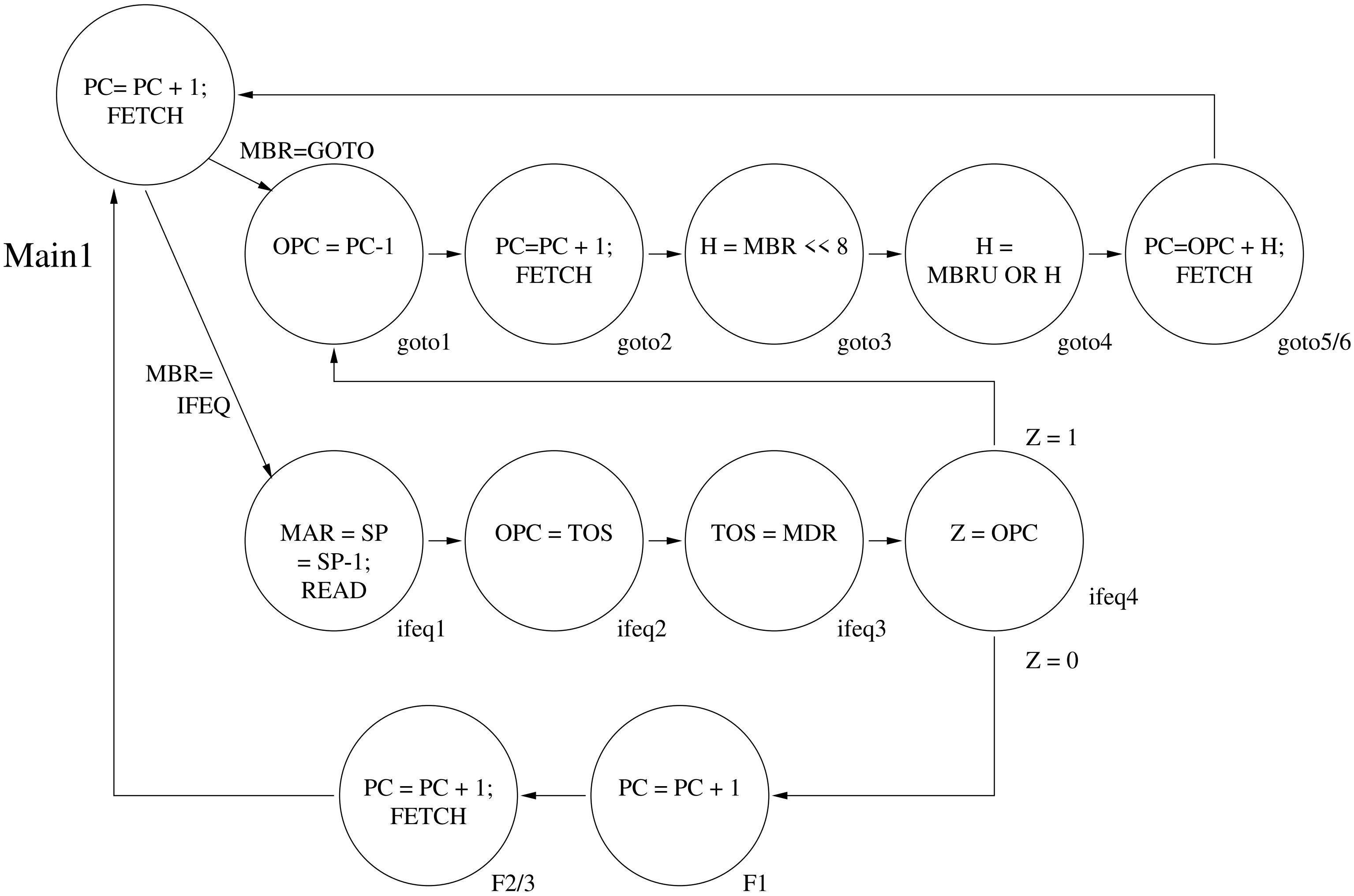
We can think of the previous four steps as four control **states**. Each state drives the control lines in specific ways.



State transitions can be conditional on contents of MBR. Note that MBR is available to control transition out of state **immediately following** state which did FETCH.

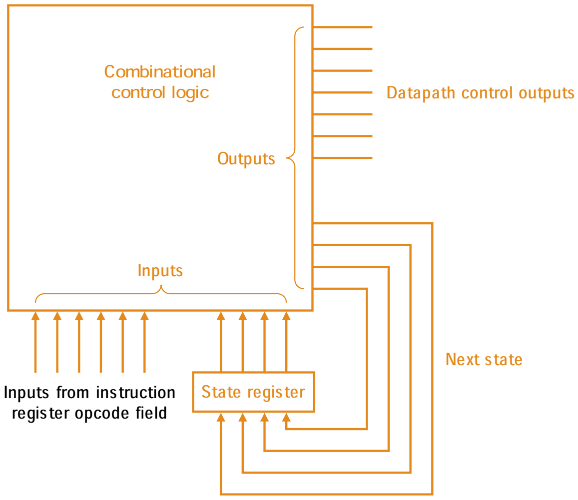
Can also be conditional on N or Z signals, which are available to control transition out of state that did ALU operation.

Jumps and Branches



FSM Implementation

Finite state machine can be implemented as a straightforward block of **combinational logic**:



Microprogrammed Instructions

Steps are performed in sequence, so we can reasonably view them as **instructions** making up a **(micro)program**.

Each instruction tells how to assert the control lines for one machine cycle: in fact, instruction encodings are just the concatenation of the control bits, plus some sequencing information.

We can store the microprogram in a ROM and fetch one instruction per cycle to drive the data path.

Most instructions have a single predecessor and successor, so we could just store them in sequence.

But how do we handle flow to a shared states (like `Main0`)?

Answer: **Every** instruction carries the 9-bit address of its successor in the microprogram ROM (i.e., every instruction ends with a goto!)

Opcode dispatch

To dispatch based on opcode MBR, we set `JMPC` with next address field = 0.

We **must** locate first microinstruction for opcode at address=opcode!

E.g., the opcode for `IADD1` is 0x60, so `iadd1` must live there.

(Subsequent instructions can be anywhere, since we can do an arbitrary `goto` after each one.)

IMPORTANT: Value of MBR is available to control next instruction **earlier** than it is available as an operand!

Example: Cycle i instruction does fetch. Cycle $i + 1$ instruction uses MBR as source for ALU operation and also sets `JMPC`. Value of MBR used for ALU operation is **old** value (before fetch), but value used for `JMPC` is **new** value (after fetch!).

Tanenbaum forgets this from time to time! Bugs:

- goto5 and goto6 can be combined (Last sentence of last full paragraph on p. 241 is false.)
- F2 and F3 can be combined
- fetch in T is superfluous

Microprogram conditional Flow

How do we handle **conditional** flow based on MBR, or N/Z ?

Answer: Special bits can be set in instruction to alter the address field before transferring control:

`JMPC`: Next address is modified by ORing MBR. (Almost always used with next address field = 0.)

`JAMZ`: Next address is modified by ORing Z bit into high-order bit. (Only useful if next address field < 0x100.)

`JAMN`: Next address is modified by ORing N bit into high-order bit. (Only useful if next address field < 0x100.)

To choose next microinstruction based on N/Z , we must set up two possible targets, at address $A < 0x100$ and $0x100 + A$, set A in the next address field, and set the `JAMN/Z` bit.

Then control will transfer to A is N/Z is clear and to $0x100 + A$ if N/Z is set.

Sometimes need to introduce an extra instruction (and hence cycle!) to make this work out, e.g., `T` and `F` targets are needed at end of conditional branch code, even though `T` could really be a synonym for `goto1`.

Assembler Syntax for Control

The `mal` syntax sweeps a lot of the microprogram control flow under the rug.

This makes it easier to read, but harder to understand!

• If no explicit transfer information appears, control passes to the next microinstruction for the current opcode, e.g., from `iadd1` to `iadd2`.

– These might not be in consecutive locations; the assembler can put them anywhere it likes, by filling the next address field appropriately.

• An auxiliary table (not shown in book) specifies fixed addresses for the first word of each opcode instruction sequence, based on the opcode values.

• Explicit control transfer can be indicated by appending a `goto` to the instruction, e.g.

```
iadd3:  MDR = TOS = MDR - H; wr; goto Main1
```

• `JMPC` is specified by using `MBR` in the `goto` target, e.g.,

```
Main1:  PC = PC + 1; fetch; goto (MBR)
```

• `JAMN/Z` is specified by using a conditional, e.g.,

```
ifeq4:  Z = OPC; if (Z) goto T; else goto F
```

– In this case, the assembler arranges to put `T` at an address $0x100$ above `F`.