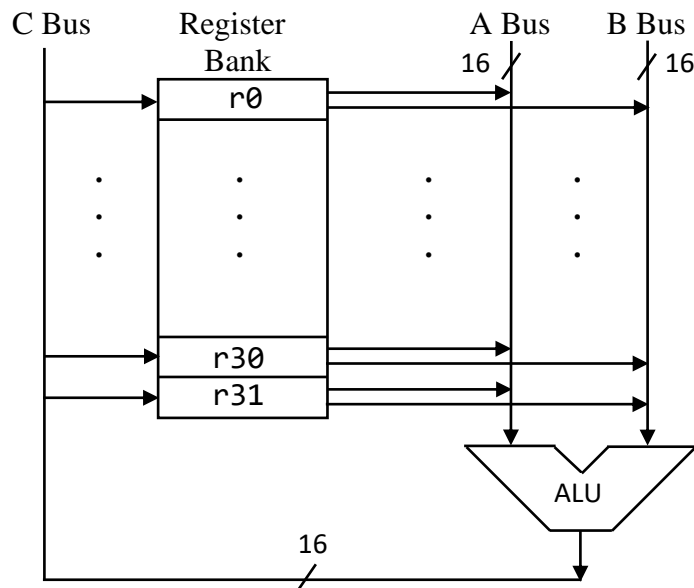
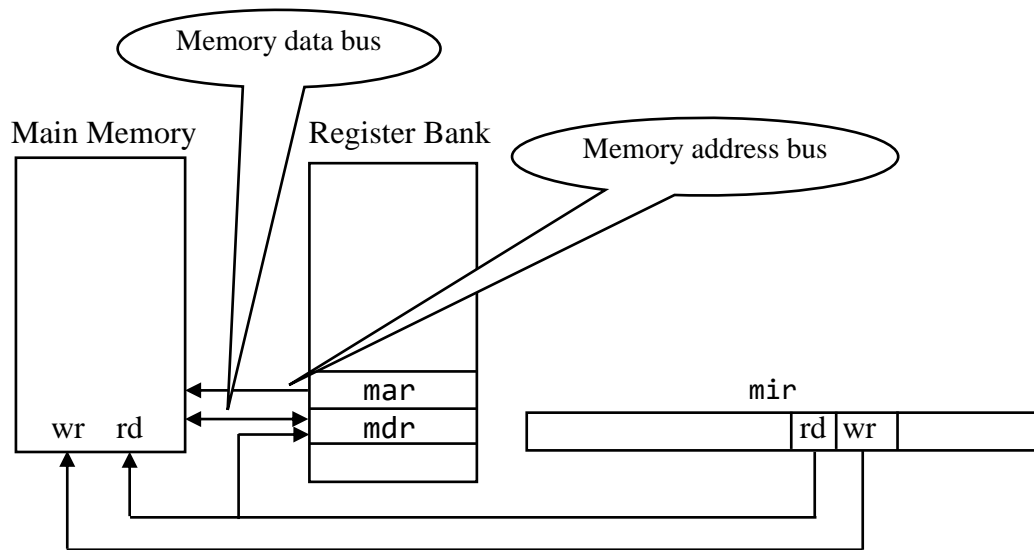


17 Microlevel of the LCC

Data Path



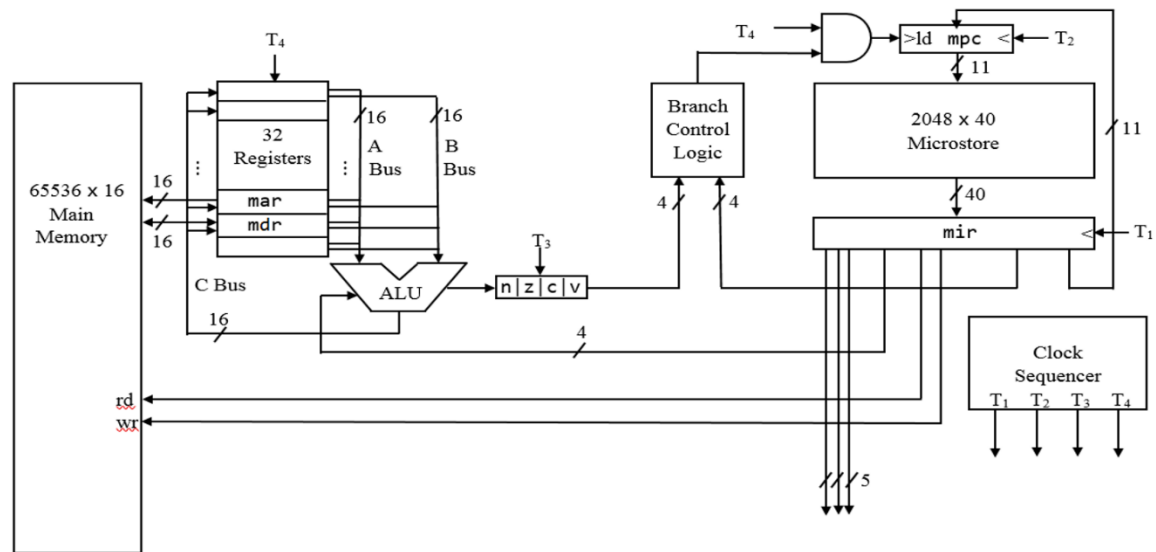
Main Memory Interface



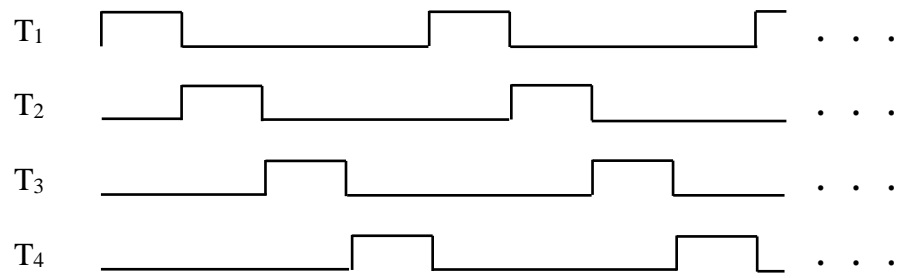
Specifying the ALU Operation



Complete Microlevel Structure of the LCC



Clock Sequencer



Binary and Symbolic Microcode

```
a.pc nop c.mar          ; load mar with contents of pc  
a.pc add b.1 c.pc rd    ; read from address in mar and incr pc
```

A binary microinstruction consists of several fields, some of which are the a, b, c, alu, and rd fields:

a	b	c	alu		rd	
---	---	---	-----	--	----	--

Microinstruction Format

A	amux	B	bmux	C	cmux	ALU	u	rd	wr	cond	addr	width
5	1	5	1	5	1	4	1	1	1	4	11	

Field

A Specifies register that inputs to the A multiplexer

amux Controls A multiplexer:
 amux = 0 then A field in *mir* drives A decoder
 amux = 1 then A field in *ir* drives A decoder

B Specifies register that inputs to the B multiplexer

bmux Controls B multiplexer:
 bmux = 0 then B field in *mir* drives B decoder
 bmux = 1 then B field in *ir* drives B decoder

C Specifies register that inputs to the C multiplexer

cmux Controls C multiplexer:
 cmux = 0 then C field in *mir* drives C decoder
 cmux = 1 then C field in *ir* drives C decoder

alu Specifies ALU operation:

F ₃	F ₂	F ₁	F ₀	Mnemonic	Output	Flags Set
0	0	0	0 (0)	nop	left	
0	0	0	1 (1)	not	~left	nz
0	0	1	0 (2)	and	left & right	nz
0	0	1	1 (3)	sext	left sign ext, (rt = mask)	nz
0	1	0	0 (4)	add	left + right	nzcv
0	1	0	1 (5)	sub	left – right	nzcv
0	1	1	0 (6)	mul	left * right	nz
0	1	1	1 (7)	div	left / right	nz
1	0	0	0 (8)	rem	left % right	nz
1	0	0	1 (9)	or	left right	nz
1	0	1	0 (10)	xor	left ^ right	nz
1	0	1	1 (11)	sll	left << right (logical)	nzc
1	1	0	0 (12)	srl	left >> right (logical)	nzc
1	1	0	1 (13)	sra	left >> right (arithmetic)	nzc
1	1	1	0 (14)	rol	left << right (rotate)	nzc
1	1	1	1 (15)	ror	left >> right (rotate)	nzc

u User n, z, c, v flags used instead of system flags

rd Initiates memory read from address in mar
wr Initiates memory write of data in mdr to address in mar

cond Specifies branch condition:

	Mnemonic	Branch if	Branch on
0	nobr		never
1	zer	$z = 1$	zero or equal
2	!zer	$z = 0$	not zero or not equal
3	neg	$n = 1$	negative
4	!neg	$n = 0$	not negative
5	cy or <	$c = 1$	less than (unsigned cmp)
6	!cy or >=	$c = 0$	grt or eq (unsigned cmp/overflow)
7	v	$v = 1$	signed overflow
8	pos	$n = z$	positive
9	lt	$n \neq v$	less than (signed cmp)
10	le	$n \neq v$ or $z = 1$	less than or equal (signed cmp)
11	gt	$n = v$ and $z = 0$	greater (signed cmp)
12	ge	$n = v$	greater than or equal (signed cmp)
13	<=	$c = 1$ or $z = 1$	less than or equal (unsigned cmp)
14	>	$c = 0$ and $z = 0$	greater than (unsigned cmp)
15	br		always

addr branch-to address

In the enhanced LCC (see Chapter 10 in “Constructing a Microprogrammed Computer Second Edition”),

- each microinstruction has a mmux bit in the rightmost position that controls the mpc multiplexer
- the mpc is 12 bits wide
- the addr field in each microinstruction is 12 bits wide
- microstore can hold up to 4096 microinstructions.

Register Names

Reg Num	Name	Initial Contents	Function
0	r0 or ac		accumulator register
1	r1		
2	r2		
3	r3		
4	r4		
5	r5 or fp		frame pointer register
6	r6 or sp		stack pointer register
7	r7 or lr		link register
8	r8 or 3	0x0003	
9	r9 or 4	0x0004	
10	r10 or 5	0x0005	
11	r11 or omask	0xf000	opcode mask
12	r12 or cmask	0x01e0	count mask (for shifts)
13	r13 or bit5	0x0020	
14	r14 or bit11	0x0800	
15	r15 or bit15	0x8000	
16	r16 or m3	0x0007	
17	r17 or m4	0x000f	
18	r18 or m5	0x001f	
19	r19 or m6	0x003f	
20	r20 or m8	0x00ff	
21	r21 or m9	0x01ff	
22	r22 or m11	0x07ff	
23	r23 or m12	0x0fff	
24	r24 or ir		machine instruction register
25	r25 or pc		program counter register
26	r26 or temp		
27	r27 or dc		decoding register
28	r28 or 1	0x0001	constant 1
29	r29 or mar		memory address register
30	r30 or mdr		memory data register
31	r31 or 0	0x0000	constant 0 (read-only register)

```

; Microcode for Tiny Instruction Set      t.sm
;=====
; Fetch machine instruction, increment pc
fetch: a.pc c.mar
      a.pc add b.1 c.pc rd      ; sim handles trap instructions here
;=====
; Decode instruction
      a.mdr add b.0 c.ir neg@ld  ; mov mdr to ir, set flags, br neg
      a.ir sll b.1 c.dc neg@add  ; sh left 1 pos, set flags, br neg
      a.dc sll b.1 c.dc neg@sub  ; sh left 1 pos, set flags, br neg
      br@ldi                    ; first 3 bits 0 so ldi inst
;=====
; Interpret machine instruction
;=====
ld:   a.ir and b.m12 c.mar      ; get x field, move to mar
      rd                      ; read from main memory
      a.mdr nop c.r0 br@fetch  ; move mdr to r0, br to fetch
;=====
add:  a.ir and b.m12 c.mar      ; get x field, move to mar
      rd                      ; read from main memory
      a.r0 add b.mdr c.r0 br@fetch ; add mdr to r0, br to fetch
;=====
sub:  ; microcode missing here
;=====
ldi:  ; microcode missing here

```

Assembling Microcode

micro t.sm

(assembles microcode to t.m)

tiny ttest.a

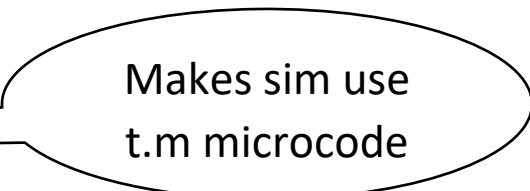
(assembles ttest.a program to ttest.e)

sim ttest.e

(runs ttest.e program using t.m)

x

Header



Makes sim use
t.m microcode

t
A 0000
A 0003
A 0006
C

Loc	Code	Source Code
		; test program for t.sm microcode
ttest		
0000	800d	ld a ; load r0 from a
0001	f002	dout ; display r0 (displays 2)
0002	f001	nl ; move cursor to next line
0003	400e	add b ; add word at b to r0
0004	f002	dout ; display r0 (displays 5)
0005	f001	nl ; move cursor to next line
0006	200d	sub a ; subtract word at a from r0
0007	f002	dout ; display r0 (displays 3)
0008	f001	nl ; move cursor to next line
0009	1007	ldi 7 ; load immediate value 7 into r0
000a	f002	dout ; display r0 (displays 7)
000b	f001	nl ; move cursor to next line
000c	f000	halt
000d	0002 a:	.word 2
000e	0003 b:	.word 3

Evaluation of the Instruction Set Architecture of the LCC

- The calling sequence for a function performs pushes and a pop. For that we have the `push` and `pop` instructions that work with the `sp` register. We also have the `sub` instruction to allocate local variables on the stack by subtracting a positive number from the `sp` register and the `add` instruction to remove items from the stack by adding a positive number to the `sp` register. For example, to allocate a word on the stack, we use

```
sub sp, sp, 1
```

To deallocate three parameters that were previously pushed onto the stack by the calling sequence of a function, we use

```
add sp, sp, 3
```

- To access parameters and dynamic local variables on the stack, we have the `fp` register that provides a fixed base address for the `ldr` and `str` instructions. For example, to load `r0` with the first parameter in a function (which has an offset of 2 with respect to the base address in the `fp` register), we use

```
ldr r0, fp, 2
```

- Getting the address of variables is easy. For global and static local variables, we have the `lea` instruction. For parameters and dynamic local variables, we have the `add` instruction. For example, to load the address of the global variable `x` into `r0`, we use

```
lea r0, x
```

To load the address of a local variable at the offset -2, we use

```
add r0, fp, -2
```

- Dereferencing pointers is easy. We use a `ld` or `ldr` instruction to get the pointer and then a `ldr` or `str` to dereference the pointer. For example, to load `r0` with the word pointed to by the local variable whose offset is -1, we use

```
ldr r0, fp, -1    ; get the pointer  
ldr r0, r0, 0     ; dereference the pointer
```

- Accessing the fields of a `struct` is easy. We can easily access any of its fields by specifying the appropriate offset in a `ldr` instruction. For example, if `r0` contains the address of a struct, to load `r1` with the field at offset 5, we use

```
ldr r1, r0, 5      ; load r1 with field at offset 5
```

- Calling a function both via the address of the function or via its name is easy. For example, to call the function whose address is in `r0`, we use

`blr r0`

To call the `f` function by name, we use

`bl f`

- We can zero out, set to 1, flip, and test individual bits within a word using the bitwise instructions (`and`, `or`, and `xor`).