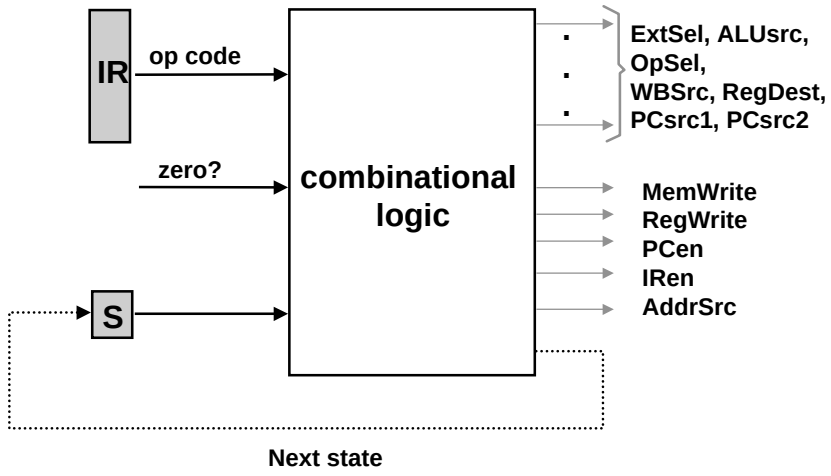
Microprogramming

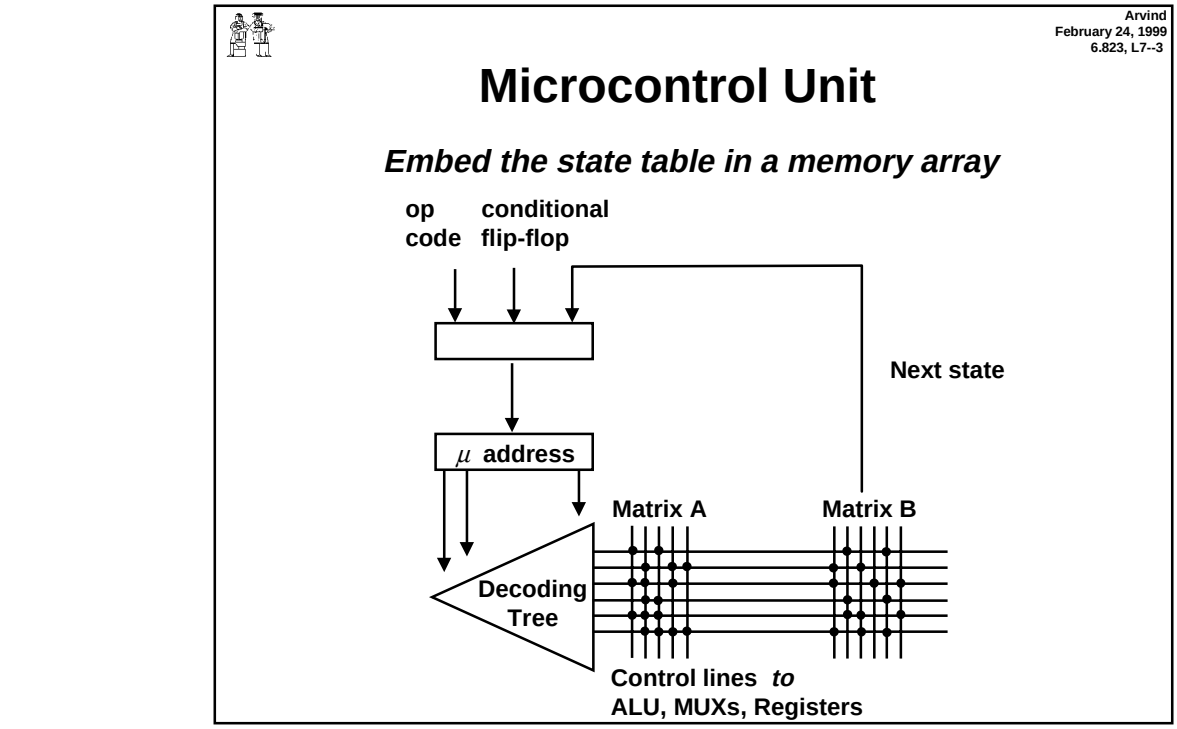
Arvind
Laboratory for Computer Science
M.I.T.

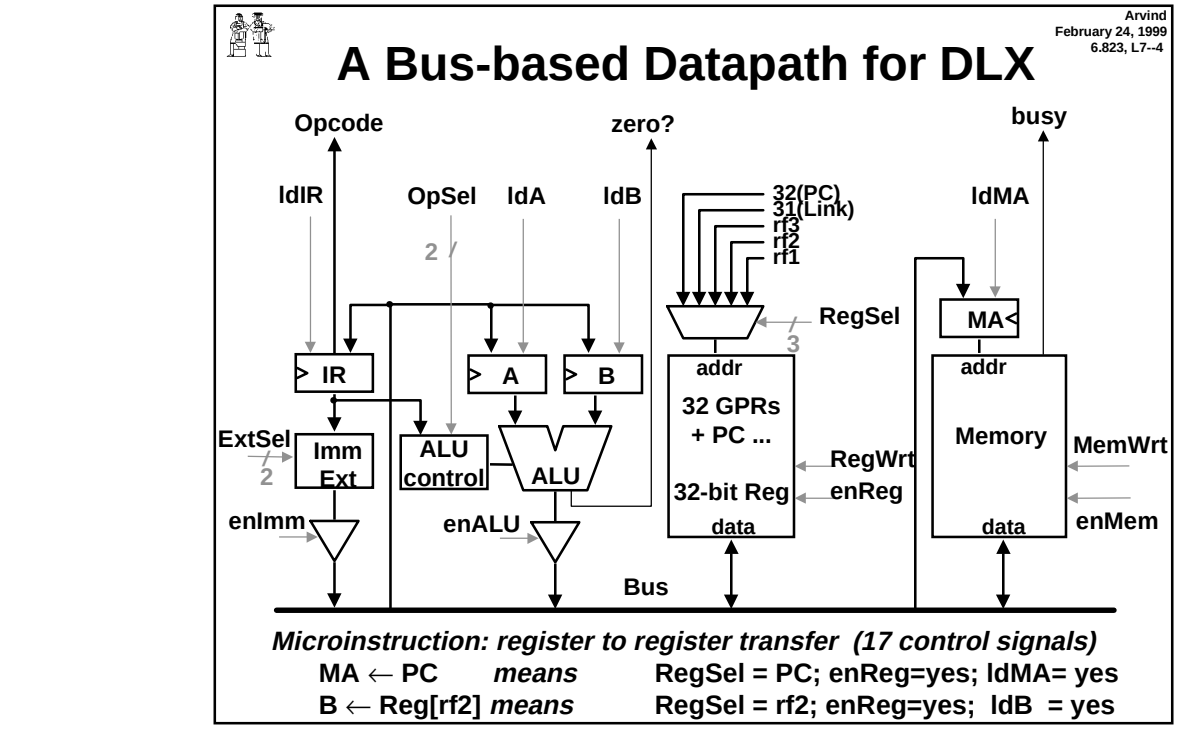
http://www.csg.lcs.mit.edu/6.823

Hardwired Controller:

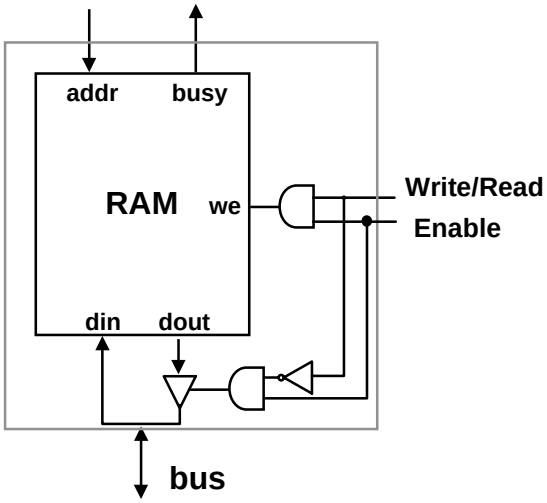
Princeton Architecture




Arvind
February 24, 1999
6.823, L7--3
Microcontrol Unit
Embed the state table in a memory array
op code
conditional flip-flop
Next state
μ address
Matrix A
Matrix B
Decoding Tree
Control lines to ALU, MUXs, Registers


Arvind
February 24, 1999
6.823, L7--4
A Bus-based Datapath for DLX
Opcode
zero?
busy
ldIR
OpSel
ldA
ldB
ldMA
32(PC)
31(Link)
rf3
rf2
rf1
RegSel
MA
IR
A
B
addr
32 GPRs + PC ...
32-bit Reg
data
addr
Memory
data
RegWrt
enReg
MemWrt
enMem
ExtSel
Imm Ext
ALU control
ALU
enImm
enALU
Bus
Microinstruction: register to register transfer (17 control signals)
MA ← PC means RegSel = PC; enReg=yes; ldMA= yes
B ← Reg[rf2] means RegSel = rf2; enReg=yes; ldB = yes

Memory Module



We will assume that Memory operates asynchronously and is slow as compared to Reg-to-Reg transfers

Microprogram Fragments

instr fetch:
MA ← PC
IR ← Memory
A ← PC
PC ← A + 4
dispatch on OPcode

can be treated as a macro

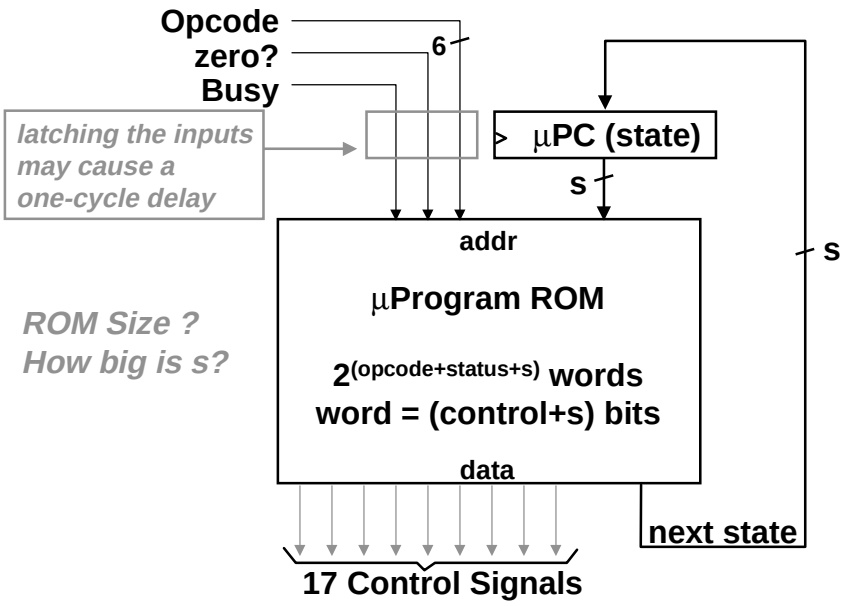
ALU:
A ← Reg[rf1]
B ← Reg[rf2]
Reg[rf3] ← func(A,B)
***do* instruction fetch**

ALUi:
A ← Reg[rf1]
B ← Imm ***sign extention ...***
Reg[rf2] ← Opcode(A,B)
***do* instruction fetch**

Microprogram Fragments *(cont.)*

LW:	**A ← Reg[rf1]**
	B ← Imm
	MA ← A + B
	Reg[rf2] ← Memory
	***do* instruction fetch**
J:	**A ← PC**
	B ← Imm
	PC ← A + B ***delay slot???***
	***do* instruction fetch**
beqz:	**A ← Reg[rf1]**
	***If* zero?(A) *then go to* bz-taken**
	***do* instruction fetch**
bz-taken:	**A ← PC**
	B ← Imm
	PC ← A + B ***delay slot???***
	***do* instruction fetch**

DLX Microcontroller: *first attempt*



Microprogram in the ROM

worksheet

State	Op	zero?	busy	Control points	next-state
$fetch_0$	*	*	*	MA ← PC	$fetch_1$
$fetch_1$	*	*	yes	....	$fetch_1$
$fetch_1$	*	*	no	IR ← Memory	$fetch_2$
$fetch_2$	*	*	*	A ← PC	$fetch_3$
$fetch_3$	*	*	*	PC ← A + 4	?
ALU_0	*	*	*	A ← Reg[rf1]	ALU_1
ALU_1	*	*	*	B ← Reg[rf2]	ALU_2
ALU_2	*	*	*	Reg[rf3]← func(A,B)	$fetch_0$

Microprogram in the ROM

State	Op	zero?	busy	Control points	next-state
$fetch_0$	*	*	*	MA ← PC	$fetch_1$
$fetch_1$	*	*	yes	....	$fetch_1$
$fetch_1$	*	*	no	IR ← Memory	$fetch_2$
$fetch_2$	*	*	*	A ← PC	$fetch_3$
$fetch_3$	ALU	*	*	PC ← A + 4	ALU_0
$fetch_3$	ALUi	*	*	PC ← A + 4	$ALUi_0$
$fetch_3$	LW	*	*	PC ← A + 4	LW_0
$fetch_3$	SW	*	*	PC ← A + 4	SW_0
$fetch_3$	J	*	*	PC ← A + 4	J_0
$fetch_3$	JAL	*	*	PC ← A + 4	JAL_0
$fetch_3$	JR	*	*	PC ← A + 4	JR_0
$fetch_3$	JALR	*	*	PC ← A + 4	$JALR_0$
$fetch_3$	beqz	*	*	PC ← A + 4	$beqz_0$
...					
ALU_0	*	*	*	A ← Reg[rf1]	ALU_1
ALU_1	*	*	*	B ← Reg[rf2]	ALU_2
ALU_2	*	*	*	Reg[rf3]← func(A,B)	$fetch_0$

Microprogram in the ROM *Cont.*

worksheet

State	Op	zero?	busy	Control points	next-state
$ALUi_0$	*	*	*	A ← Reg[rf1]	$ALUi_1$
$ALUi_1$	sExt	*	*	B ← $sExt_{16}$(Imm)	$ALUi_2$
$ALUi_1$	uExt	*	*	B ← $uExt_{16}$(Imm)	$ALUi_2$
$ALUi_2$	*	*	*	Reg[rf3]← Op(A,B)	$fetch_0$
...					
J_0	*	*	*	A ← PC	J_1
...					
$beqz_0$	*	*	*	A ← Reg[rf1]	$beqz_1$
...					

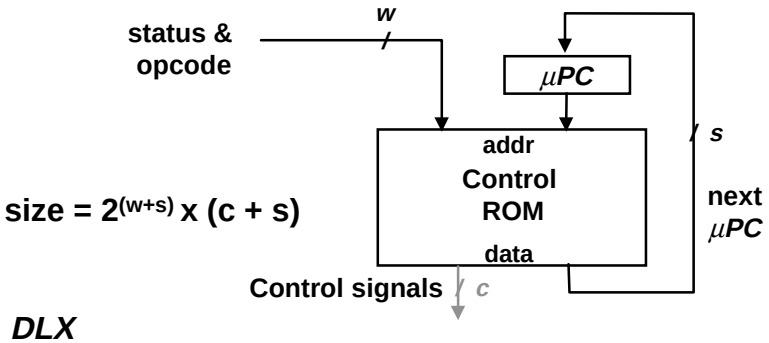
no delay slots

Microprogram in the ROM *Cont.*

State	Op	zero?	busy	Control points	next-state
$ALUi_0$	*	*	*	A ← Reg[rf1]	$ALUi_1$
$ALUi_1$	sExt	*	*	B ← $sExt_{16}$(Imm)	$ALUi_2$
$ALUi_1$	uExt	*	*	B ← $uExt_{16}$(Imm)	$ALUi_2$
$ALUi_2$	*	*	*	Reg[rf3]← Op(A,B)	$fetch_0$
...					
J_0	*	*	*	A ← PC	J_1
J_1	*	*	*	B ← $sExt_{26}$(Imm)	J_2
J_2	*	*	*	PC ← A+B	$fetch_0$
...					
$beqz_0$	*	*	*	A ← Reg[rf1]	$beqz_1$
$beqz_1$	*	yes	*	A ← PC	$beqz_2$
$beqz_1$	*	no	*	....	$fetch_0$
$beqz_2$	*	*	*	B ← $sExt_{16}$(Imm)	$beqz_3$
$beqz_3$	*	*	*	PC ← A+B	$fetch_0$
...					

no delay slots

Size of Control Store



size = $2^{(w+s)}$ x (c + s)

DLX

w = 6+2 c = 17 s = ?

no. of steps per opcode = 4 to 6 + fetch-sequence

no. of states ≈ (4 steps per op-group) x op-groups
+ common sequences

= 4 x 8 + 10 states = 42 states

⇒ s = 6

Control ROM = $2^{(8+6)}$ x 23 bits ≈ 48 k bytes

Reducing the Size of Control Store

Control store has to be *fast* ⇒ *expensive*

Reduce the ROM height (= address bits)

⇒ *reduce inputs by extra external logic*
each input bit doubles the size of the control store

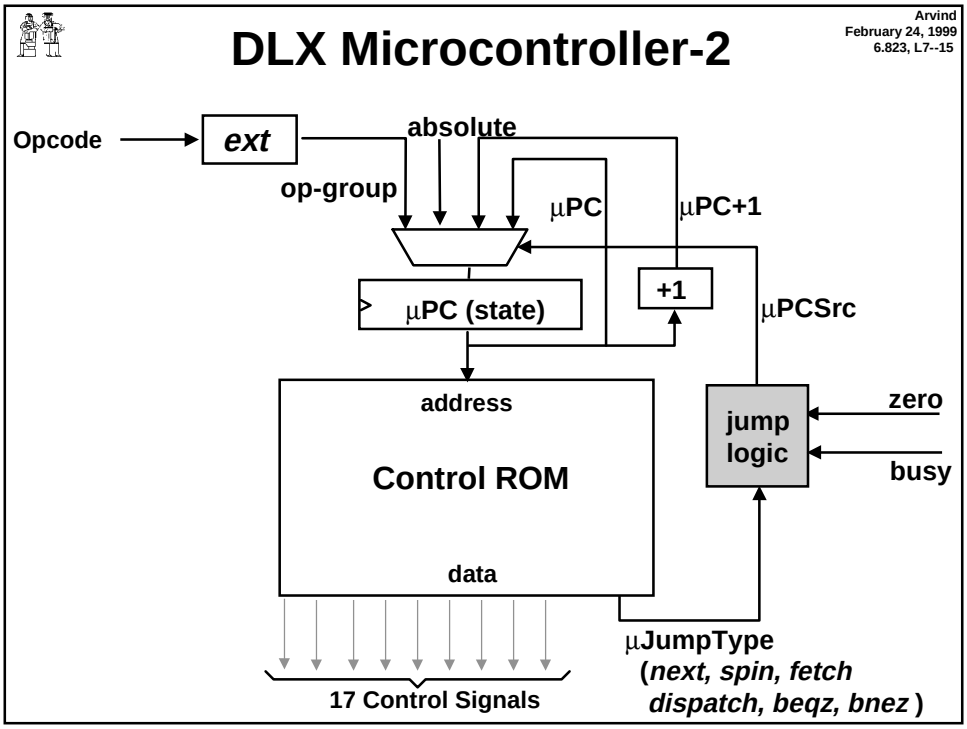
⇒ *reduce states by grouping opcodes*
find common sequences of actions

⇒ *condense input status bits*
combine all exceptions into one, i.e., exception/no-exception

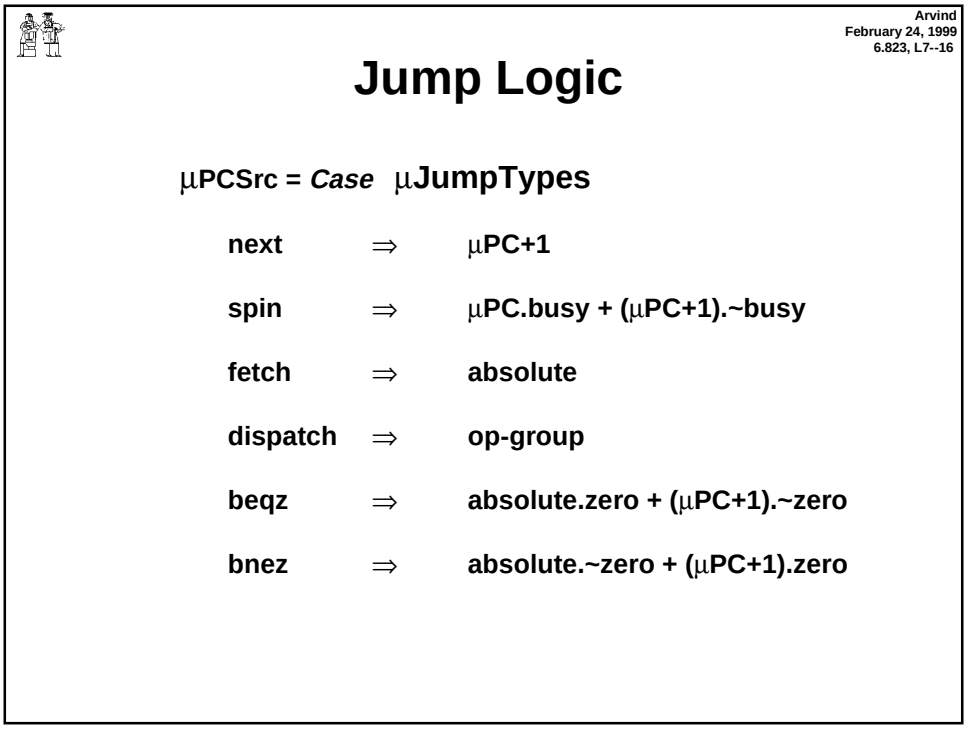
Reduce the ROM width

⇒ *restrict the next-state encoding*
Next, Dispatch on opcode, Wait for memory, ...

⇒ *encode control signals*



DLX Microcontroller-2
Arvind
February 24, 1999
6.823, L7--15
Opcode
ext
absolute
op-group
μPC
μPC+1
+1
μPC (state)
μPCSrc
address
Control ROM
data
jump
logic
zero
busy
μJumpType
(next, spin, fetch
dispatch, beqz, bnez)
17 Control Signals



Arvind
February 24, 1999
6.823, L7--16
Jump Logic
μPCSrc = Case μJumpTypes
next ⇒ μPC+1
spin ⇒ μPC.busy + (μPC+1).~busy
fetch ⇒ absolute
dispatch ⇒ op-group
beqz ⇒ absolute.zero + (μPC+1).~zero
bnez ⇒ absolute.~zero + (μPC+1).zero

DLX-Controller-2 *worksheet*

State	Control points	next-state
$fetch_0$	MA ← PC	
$fetch_1$	IR ← Memory	
$fetch_2$	A ← PC	
$fetch_3$	PC ← A + 4	
...		
ALU_0	A ← Reg[rf1]	
ALU_1	B ← Reg[rf2]	
ALU_2	Reg[rf3]← func(A,B)	
...		
J_0	A ← PC	
J_1	B ← $sExt_{26}$(Imm)	
J_2	PC ← A+B	
...		
$BEQZ_0$	A ← Reg[rf1]	
$BEQZ_1$		
$BEQZ_2$	A ← PC	
$BEQZ_3$	B ← $sExt_{16}$(Imm)	
$BEQZ_4$	PC ← A+B	

Instruction Fetch & ALU: *DLX-Controller-2*

State	Control points	next-state
$fetch_0$	MA ← PC	next
$fetch_1$	IR ← Memory	spin
$fetch_2$	A ← PC	next
$fetch_3$	PC ← A + 4	dispatch
...		
ALU_0	A ← Reg[rf1]	next
ALU_1	B ← Reg[rf2]	next
ALU_2	Reg[rf3]← func(A,B)	fetch
$ALUi_0$	A ← Reg[rf1]	next
$ALUi_1$	B ← $sExt_{16}$(Imm)	next
$ALUi_2$	Reg[rf3]← Op(A,B)	fetch
$ALUui_0$	A ← Reg[rf1]	next
$ALUui_1$	B ← $uExt_{16}$(Imm)	next
$ALUui_2$	Reg[rf3]← Op(A,B)	fetch

Load & Store: *DLX-Controller-2*

State	Control points	next-state
LW_0	A ← Reg[rf1]	next
LW_1	B ← $sExt_{16}$(Imm)	next
LW_2	MA ← A+B	next
LW_3	Reg[rf2] ← Memory	spin
LW_4		fetch
SW_0	A ← Reg[rf1]	next
SW_1	B ← $sExt_{16}$(Imm)	next
SW_2	MA ← A+B	next
SW_3	Memory ← Reg[rf2]	spin
SW_4		fetch

Branches: *DLX-Controller-2*

State	Control points	next-state
$BEQZ_0$	A ← Reg[rf1]	next
$BEQZ_1$		bnez
$BEQZ_2$	A ← PC	next
$BEQZ_3$	B ← $sExt_{16}$(Imm)	next
$BEQZ_4$	PC ← A+B	fetch
$BNEZ_0$	A ← Reg[rf1]	next
$BNEZ_1$		beqz
$BNEZ_2$	A ← PC	next
$BNEZ_3$	B ← $sExt_{16}$(Imm)	next
$BNEZ_4$	PC ← A+B	fetch

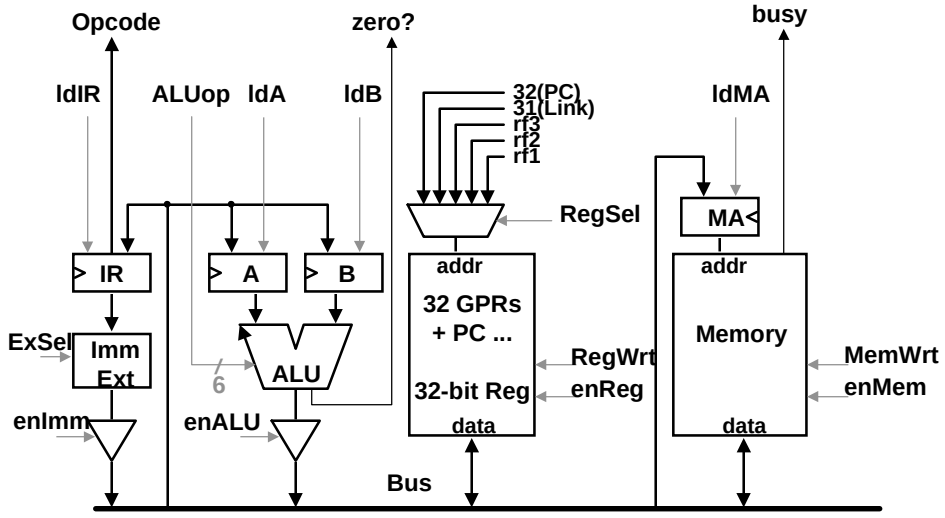
no delay slots

Jumps: *DLX-Controller-2*

State	Control points	next-state
J_0	A ← PC	next
J_1	B ← $sExt_{26}$(Imm)	next
J_2	PC ← A+B	fetch
JR_0	PC ←Reg[rf1]	fetch
JAL_0	Reg[R31] ← PC	next
JAL_1	A ← PC	next
JAL_2	B ← $sExt_{26}$(Imm)	next
JAL_3	PC ← A+B	fetch
$JALR_0$	Reg[R31] ← PC	next
$JALR_1$	PC ←Reg[rf1]	fetch

no delay slots

Implementing Complex Instructions



rf3 ← M[(rf1)] op (rf2)	*Reg-Memory-src ALU op*
M[(rf3)] ← (rf1) op (rf2)	*Reg-Memory-dst ALU op*
M[(rf3)] ← M[(rf1)] op M[(rf2)]	*Mem-Mem ALU op*

Reg-Mem ALU Instructions:

DLX-Controller-2

Reg-Memory-src ALU op **rf3 ← M[(rf1)] op (rf2)**

$ALUM_0$	**MA ← Reg[rf1]**	**next**
$ALUM_1$	**A ← Memory**	**spin**
$ALUM_2$	**B ← Reg[rf2]**	**next**
$ALUM_3$	**Reg[rf3] ← func(A,B)**	**fetch**
$ALUMi_0$	**MA ← Reg[rf1]**	**next**
$ALUMi_1$	**A ← Memory**	**spin**
$ALUMi_2$	**B ← $sExt_{16}$(Imm)**	**next**
$ALUMi_3$	**Reg[rf2] ← Op(A,B)**	**fetch**

Reg-Memory-dst ALU op **M[(rf3)] ← (rf1) op (rf2)**

$ALUMD_0$	**MA ← Reg[rf3]**	**next**
$ALUMD_1$	**A ← Reg[rf1]**	**next**
$ALUMD_2$	**B ← Reg[rf2]**	**next**
$ALUMD_3$	**Memeory ← func(A,B)**	**spin**
$ALUMD_3$		**fetch**

Mem-Mem ALU Instructions:

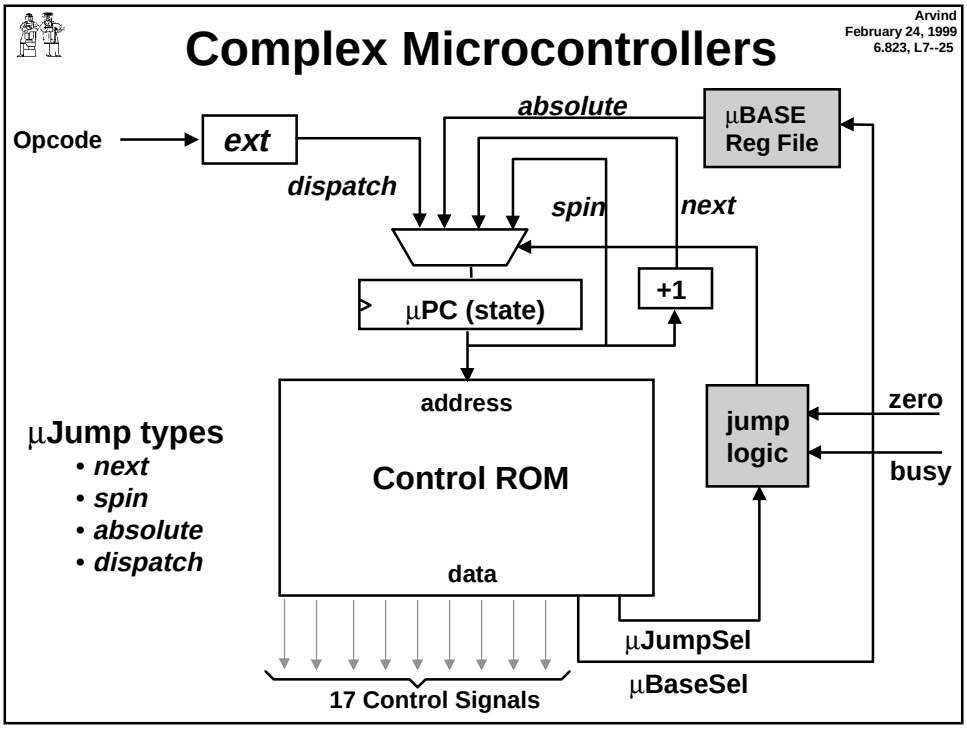
DLX-Controller-2

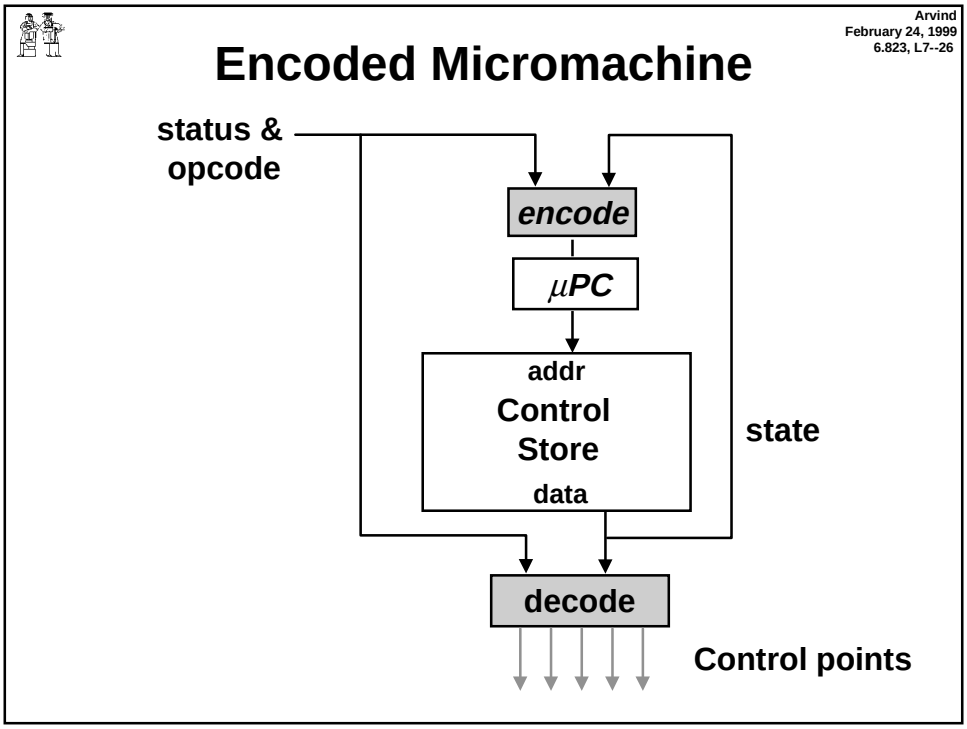
Mem-Mem ALU op **M[(rf3)] ← M[(rf1)] op M[(rf2)]**

$ALUMM_0$	**MA ← Reg[rf1]**	**next**
$ALUMM_1$	**A ← Memory**	**spin**
$ALUMM_2$	**MA ← Reg[rf2]**	**next**
$ALUMM_3$	**B ← Memory**	**spin**
$ALUMM_4$	**MA ← Reg[rf3]**	**next**
$ALUMM_5$	**Memeory ← func(A,B)**	**spin**
$ALUMM_6$		**fetch**

Complex instructions usually do not require datapath modifications in a microprogrammed implementation
-- only extra space for the control program

Implementing these instructions using a hardwired controller is difficult without datapath modifications


Arvind
February 24, 1999
6.823, L7--25
Complex Microcontrollers
Opcode
ext
absolute
μBASE
Reg File
dispatch
spin
next
+1
μPC (state)
address
Control ROM
data
jump
logic
zero
busy
μJump types
• next
• spin
• absolute
• dispatch
μJumpSel
μBaseSel
17 Control Signals


Arvind
February 24, 1999
6.823, L7--26
Encoded Micromachine
status &
opcode
encode
μPC
addr
Control
Store
data
state
decode
Control points

Performance Issues

Microprogrammed control
$\Rightarrow$ multiple cycles per instruction

Cycle time ?
$t_C > \max(t_{reg\text{-}reg}, t_{ALU}, t_{\mu ROM}, t_{RAM})$

Given complex control, t_{ALU} & t_{RAM} can be broken into multiple cycles. However, $t_{\mu ROM}$ cannot be broken down. Hence
$t_C > \max(t_{reg\text{-}reg}, t_{\mu ROM})$

Suppose $10 * t_{\mu ROM} < t_{RAM}$
good performance, relative to the single-cycle hardwired implementation, can be achieved even with a CPI of 10

Microprogramming in the Seventies

thrived because

- **Significantly faster ROMs than DRAMs were available**
- **For complex instruction sets, datapath and controller were *cheaper and simpler***
- ***New instructions* , e.g., floating point, could be supported without datapath modifications**
- ***Fixing bugs* in the controller was easier**
- **ISA compatibility across various models could be achieved cheaply**

VLSI & Microprogramming

By late seventies

- **technology assumption about ROM & RAM speed became invalid**

- **micromachines became more complicated**
  - **to overcome slower ROM, micromachines were pipelined**
  - **complex instruction sets led to the need for subroutine and call stacks in μcode.**
  - **need for fixing bugs in control programs was in conflict with read-only nature of μROM**

    B1700, QMachine, Intel432, ...
- **introduction of caches and buffers, especially for instructions, made multiple-cycle execution of reg-reg instructions unattractive**

Modern Usage

Microprogramming is far from extinct

Played a crucial role in micros of the Eighties,
Intel 386 and 486, Motorola 68K

Microprogramming is present in most micros of the Nineties in an assisting role

- **Most instructions are executed directly, i.e. without invoking any microcode**
- **Infrequently-used complex instruction (legacy codes) trap and invoke the micromachinery**

VAX-compatibility on DEC Alpha is achieved via microcoding