Microprogramming

Krste Asanovic
Laboratory for Computer Science
M.I.T.

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

The DLX ISA

Processor State

- 32 32-bit GPRs, R0 always contains a 0
- 32 single precision FPRs, may also be viewed as
  16 double precision FPRs
- FP status register, used for FP compares & exceptions
- PC, the program counter
- some other special registers

Data types

- 8-bit byte, 2-byte half word
- 32-bit word for integers
- 32-bit word for single precision floating point
- 64-bit word for double precision floating point

Load/Store style instruction set

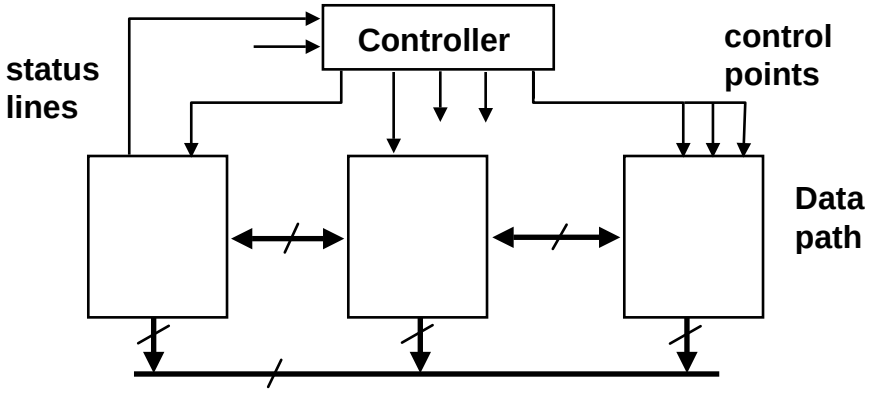
- data addressing modes- immediate & indexed
- branch addressing modes- PC relative & register indirect
- Byte addressable memory- big-endian mode

All instructions are 32 bits

(See Chapter 2, H&P for full description)

Microarchitecture

Implementation of an ISA



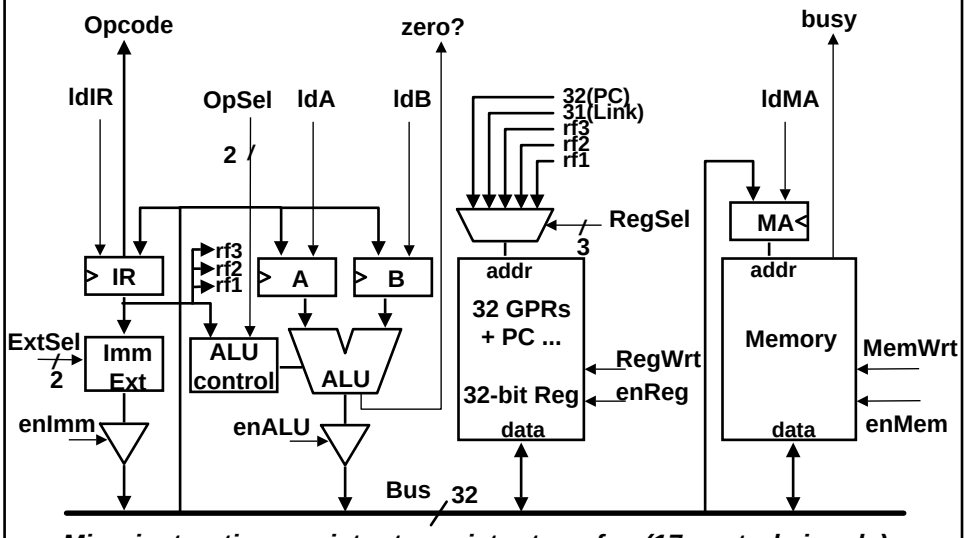
Structure: **How components are connected.**

Static

Sequencing: **How data moves between components**

Dynamic

A Bus-based Datapath for DLX

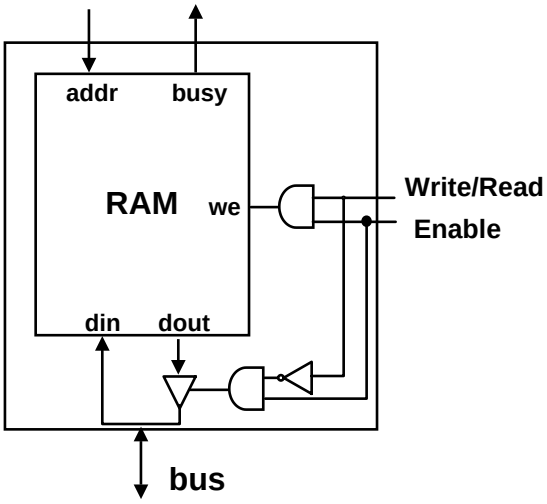


Microinstruction: register to register transfer (17 control signals)

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

Instruction Execution

Execution of a DLX instruction involves

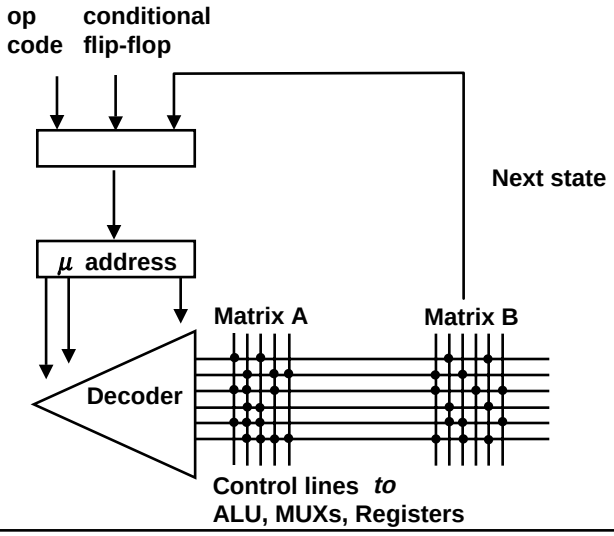
1. **instruction fetch**
2. **decode and register fetch**
3. **ALU operation**
4. **memory operation (optional)**
5. **write back to register file (optional)**

and the computation of the address of the *next instruction*

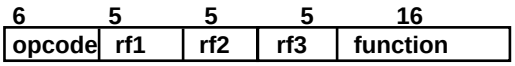
Microcontrol Unit

Maurice Wilkes, 1954

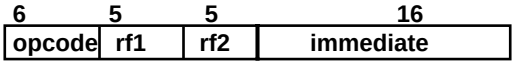
Embed the control logic state table in a memory array



DLX ALU Instructions



Register-Register form:
Reg[rf3] ← function(Reg[rf1], Reg[rf2])



Register-Immediate form:
Reg[rf2] ← function(Reg[rf1], SignExt(immediate))

Microprogram Fragments

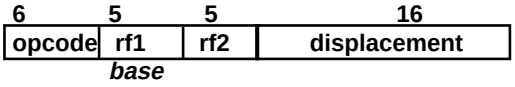
instr fetch:
MA ← PC
IR ← Memory
A ← PC
PC ← A + 4
dispatch on OPcode

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**

DLX Load/Store Instructions



Load/store byte, halfword, word to/from GPR:

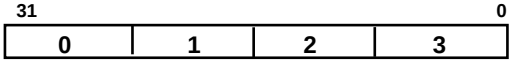
LB, LBU, SB, LH, LHU, SH, LW, SW

byte and half-word can be sign or zero extended

Load/store single and double FP to/from FPR:

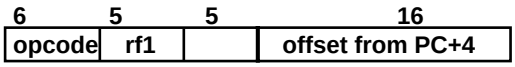
LF, LD, SF, SD

- **Byte addressable machine**
- **Memory access must be data aligned**
- **A single addressing mode**
  (base) + displacement
- **Big-endian byte ordering**



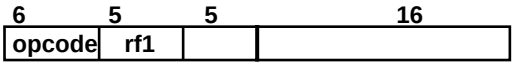
DLX Control Instructions

Conditional branch on GPR



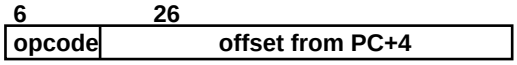
BEQZ, BNEZ

Unconditional register-indirect jumps



JR, JALR

Unconditional PC-relative jumps

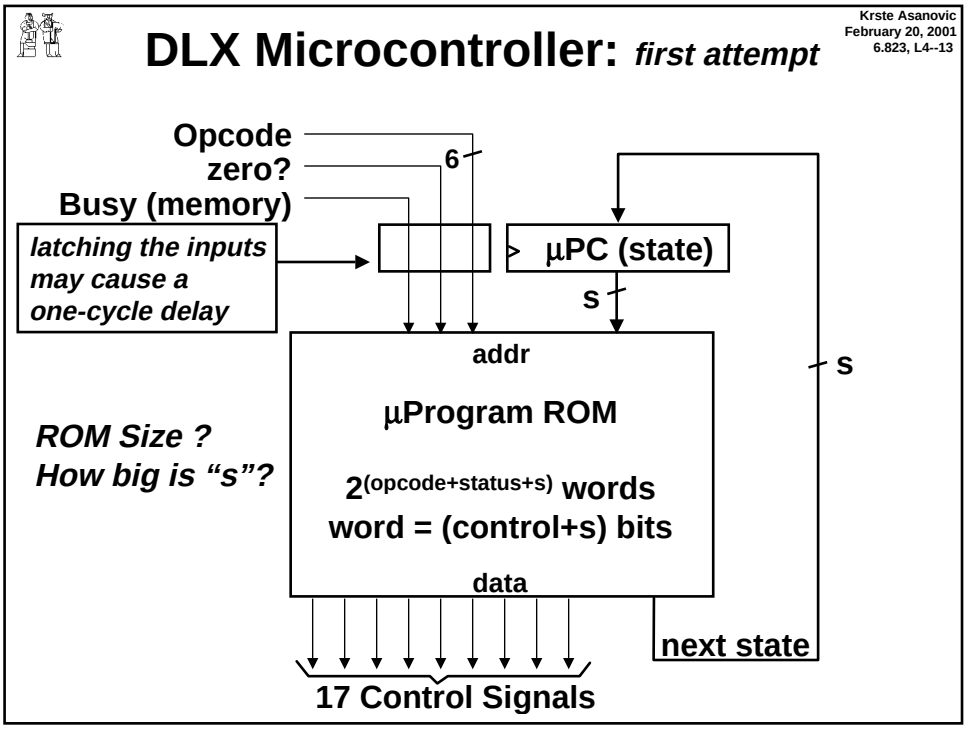


J, JAL

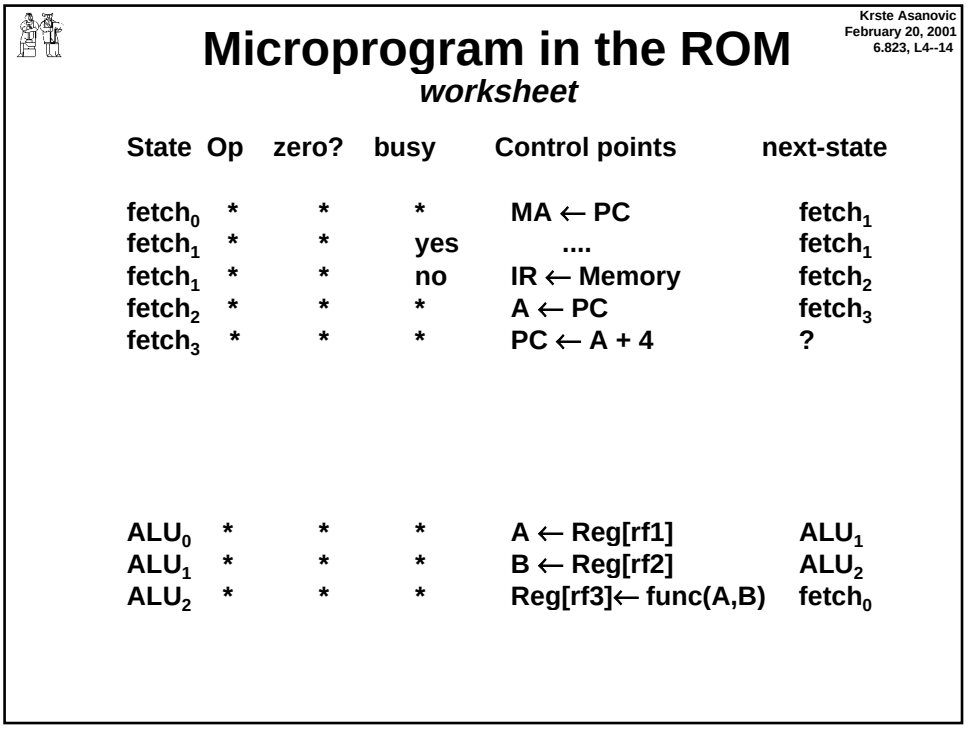
- **PC-offset are specified in bytes**
- **jump-&-link (JAL) stores PC+4 into the link register (R31)**
- ***(Real DLX has delayed branches – ignored this lecture)***

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
	***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
	***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)	$fetch_0$

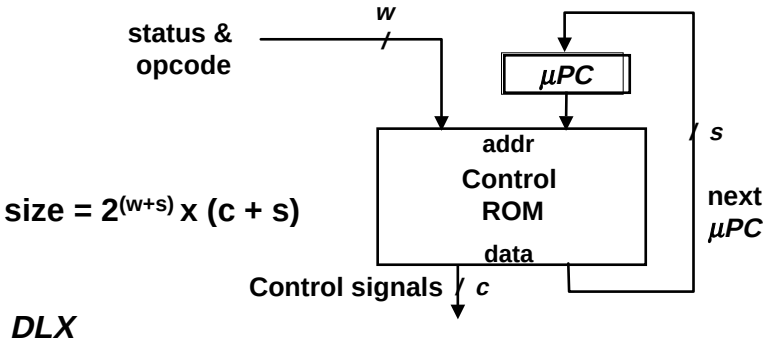
Microprogram in the ROM

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.*

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	$fetch_0$
...					

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 Kbytes

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

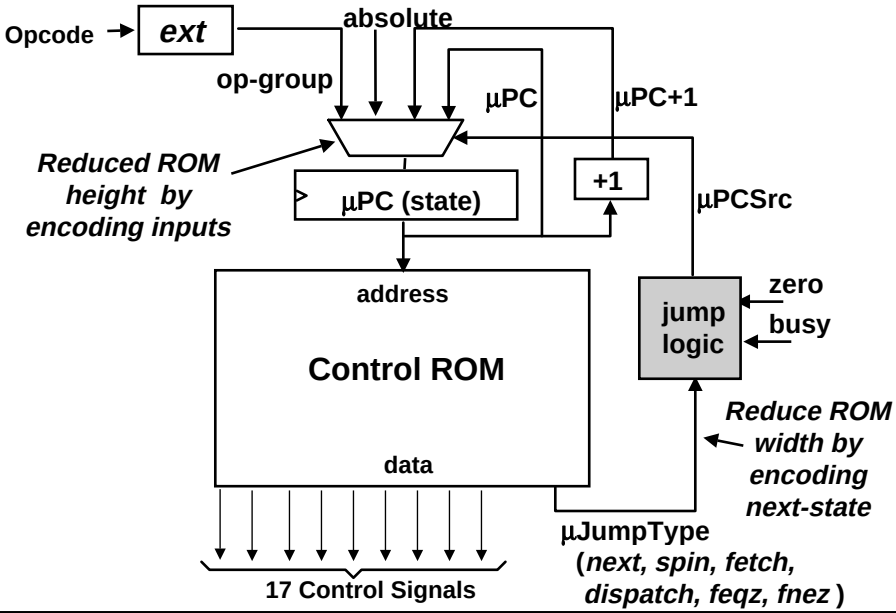
⇒ *condense input status bits*
combine all exceptions into one, i.e., exception/no-exception

Reduce the ROM width

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

⇒ *encode control signals (vertical microcode)*

DLX Microcontroller-2



Jump Logic

μPCSrc = *Case* μJumpTypes

next	⇒	μPC+1
spin	⇒	if (busy) then μPC else μPC+1
fetch	⇒	absolute
dispatch	⇒	op-group
feqz	⇒	if (zero) then absolute else μPC+1
fnez	⇒	if (zero) then μPC+1 else absolute

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

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
$BEQZ_1$		fnez
$BEQZ_2$	A ← PC	next
$BEQZ_3$	B ← $sExt_{16}$(Imm)	next
$BEQZ_4$	PC ← A+B	fetch
$BNEZ_0$	A ← Reg[rf1]	next
$BNEZ_1$		feqz
$BNEZ_2$	A ← PC	next
$BNEZ_3$	B ← $sExt_{16}$(Imm)	next
$BNEZ_4$	PC ← A+B	fetch

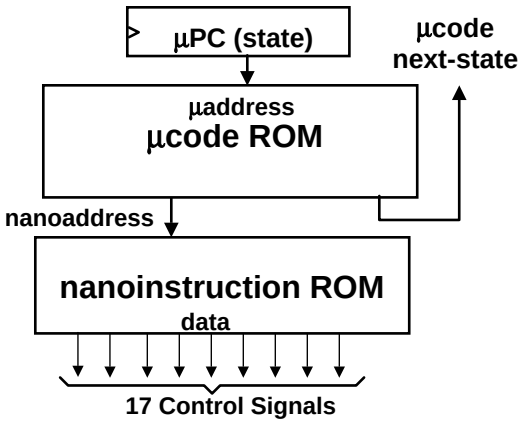
Jumps: *DLX-Controller-2*

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[31] ← PC	next
JAL_1	A ← PC	next
JAL_2	B ← $sExt_{26}$(Imm)	next
JAL_3	PC ← A+B	fetch
$JALR_0$	Reg[31] ← PC	next
$JALR_1$	PC ←Reg[rf1]	fetch

Nanocoding

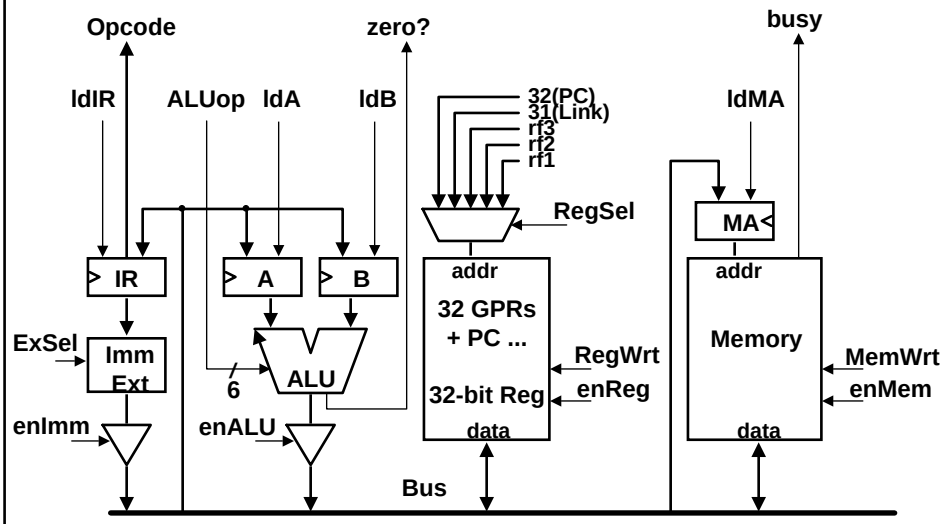
Exploits recurring control signal patterns in μcode, e.g.,

ALU_0 A ← Reg[rf1]
...
$ALUi_0$ A ← Reg[rf1]
...



- Place common control signal patterns into *nanoinstruction* table
- μcode word contains pointer to nanoinstruction
- Higher latency to get control signals out

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*

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
$ALUMM_5$	Memory ← 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

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**

Writable Control Store (WCS)

Implement control store with SRAM not ROM

- **MOS SRAM memories now almost as fast as control store (core memories/DRAMs were 10x slower)**
- **Bug-free microprograms difficult to write**

WCS provided as option on several minicomputers

- **Allowed users to change microcode for each process**

WCS *failed*

- **Little or no programming tools support**
- **Hard to fit software into small space**
- **Microcode control tailored to original ISA, less useful for others**
- **Large WCS part of processor state - expensive context switches**
- **Protection difficult if user can change microcode**
- **Virtual memory required ***restartable* microcode**

Performance Issues

Microprogrammed control
⇒ 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

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**
    ⇒ *WCS (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,
Motorola 68K (first microcoded microprocessor)
Intel 386 and 486

Microcode is present in most modern CISC micros in an assisting role *(e.g. AMD Athlon, Intel Pentium-4)*

- **Most instructions are executed directly, i.e., with hard-wired control**
- **Infrequently-used and/or complicated instructions invoke the microcode engine**

***Patchable* microcode common for post-fabrication bug fixes, e.g. Intel Pentiums load μcode patches at bootup**