Agenda

- VAX and its History
- VAX ISA
- VAX Virtual Address
- Microcode

What is VAX?

***V**irtual **A**ddress e**X**tension*

- Developed by Digital Equipment Corporation (DEC) in the mid-1970s
- A 32-bit CISC orthogonal instruction set
- A commercial pioneer in using virtual address
- Replace 16-bit PDP-11 ISA
- 15 – 20 year architecture life span
- Compatible with PDP-11 software

VAX History

First VAX [illegible] installed in CMU

1977 – VAX 11/780 TTL
1980 – VAX 11/750 TTL
1980 – VAX 11/730 TTL
1984 – VAX 8600 ECL
1985 – MicroVAX II
MicroVAX chip
1986 – VAX 8800 ECL
1987 – MicroVAX 3600
CVAX chip
VAX station 2000
1989 – VAX6600
NVAX chip
1989 – VAX9000 ECL

A single ISA with diversified and evolved hardware implementations

VAX ISA Summary

- 32-bit CISC Architecture
- 16 32-bit registers (r0, r1, .., r15)
  - r12, r13, r14, r15 reserved for AP, FP, SP, PC
- 300+ variable length instructions
- 22 addressing modes
- ISA Designed for Compiler Simplicity and Reduced Code Size

Data Types

Data Type	Size	Range (decimal)	
Integer		Signed	Unsigned
Byte	8 bits	-128 to +127	0 to 255
Word	16 bits	-32768 to +32767	0 to 65535
Longword	32 bits	-2^{31} to $+2^{31}-1$	0 to $2^{32}-1$
Quadword	64 bits	-2^{63} to $+2^{63}-1$	0 to $2^{64}-1$
Octaword	128 bits	-2^{127} to $+2^{127}-1$	0 to $2^{128}-1$
Float			
F_floating	32 bits	Approx. 7 decimal digits precision.	
D_floating	64 bits	Approx. 16 decimal digits precision.	
G_floating	64 bits	Approx. 15 decimal digits precision.	
H_floating	128 bits	Approx. 33 decimal digits precision.	
Packed Decimal String	0 to 16 bytes (31 digits)	Numeric, two digits per byte, sign in low half of last byte	
Character String	0 to 65535 bytes	One character per byte	
Variable-length Bit Field	0 to 31 bytes (DIGITS)	$-10^{31}-1$ to $+10^{31}-1$	
Queue	>= 2 longwords / queue entry	0 through 2 billion entries	

Recreated from Table 2-1 in VAX Architecture Handbook (1981)

Though VAX is a 32-bit Architecture, the instructions can operate on multiple data formats.

Note that in VAX, a *word* differs from the *word* we refer to in MIPS or other class.

Most frequently:
• Integers/Floats of varying sizes.
• Character String

VAX Literally: *word* is equivalent to one of our *halfwords*.

Rich hardware data types to simplify compiler

Sample Instructions (1/3)

Instruction type	Example	Instruction meaning
Data transfers	Move data between byte, half-word, word, or double-word operands; * is data type	
	mov*	Move between two operands
	movzb*	Move a byte to a half word or word, extending it with zeros
	mova*	Move the 32-bit address of an operand; data type is last
	push*	Push operand onto stack
Arithmetic/logical	Operations on integer or logical bytes, half words (16 bits), words (32 bits); * is data type	
	add*_	Add with 2 or 3 operands
	cmp*	Compare and set condition codes
	tst*	Compare to zero and set condition codes
	ash*	Arithmetic shift
	clr*	Clear
	cvtb*	Sign-extend byte to size of data type

Figure E.4 Classes of VAX instructions with examples. The asterisk stands for multiple data types: b, w, l, d, f, g, h, and q. The underline, as in addd_, means there are 2-operand (addd2) and 3-operand (addd3) forms of this instruction.

Sample Instructions (2/3)

Control	Conditional and unconditional branches	
	`beql, bneq`	Branch equal, branch not equal
	`bleq, bgeq`	Branch less than or equal, branch greater than or equal
	`brb, brw`	Unconditional branch with an 8-bit or 16-bit address
	`jmp`	Jump using any addressing mode to specify target
	`aobleq`	Add one to operand; branch if result ≤ second operand
	`case_`	Jump based on case selector
Procedure	Call/return from procedure	
	`calls`	Call procedure with arguments on stack (see Section E.6)
	`callg`	Call procedure with FORTRAN-style parameter list
	`jsb`	Jump to subroutine, saving return address (like MIPS `jal`)
	`ret`	Return from procedure call

Figure E.4 Classes of VAX instructions with examples. The asterisk stands for multiple data types: b, w, l, d, f, g, h, and q. The underline, as in `addd_`, means there are 2-operand (`addd2`) and 3-operand (`addd3`) forms of this instruction.

Sample Instructions (3/3)

Floating point	Floating-point operations on D, F, G, and H formats	
	`addd_`	Add double-precision D-format floating numbers
	`subd_`	Subtract double-precision D-format floating numbers
	`mulf_`	Multiply single-precision F-format floating point
	`polyf`	Evaluate a polynomial using table of coefficients in F format
Other	Special operations	
	`crc`	Calculate cyclic redundancy check
	`insque`	Insert a queue entry into a queue

Figure E.4 Classes of VAX instructions with examples. The asterisk stands for multiple data types: b, w, l, d, f, g, h, and q. The underline, as in `addd_`, means there are 2-operand (`addd2`) and 3-operand (`addd3`) forms of this instruction.

Instruction Variants

- Instruction Variants
  - Operation + Data type + # of Operands
    - Operation – add, sub …
    - Data type – byte, word, dword,
    - # of operands – 1 ~ 3 register and 1~3 memory (depends on operations)
- One Sample —— ADD

The operation add works with data types byte, word, long, float, and double and comes in versions for either 2 or 3 unique operands, so the following instructions are all found in the VAX:

```
addb2    addw2    addl2    addf2    addd2
addb3    addw3    addl3    addf3    addd3
```

The total combination of instructions is 304+, not including address mode variances!

Addressing Modes

- General Register Addressing
  - Literal (3 types)
  - Register
  - Register deferred
  - Autodecrement and Autoincrement
  - Autoincrement deferred
  - Byte, Word, Longword displacement
  - Byte, Longword displacement deferred
  - Indexed
- Program Counter Addressing
  - Immediate
  - Absolute
  - Byte relative
  - Byte relative deferred
  - Word relative
  - Word relative deferred
  - Longword relative
  - Longword relative deferred

Register – Register Register–Memory Memory – Memory

Total 22 Addressing Modes

Address mode syntax examples

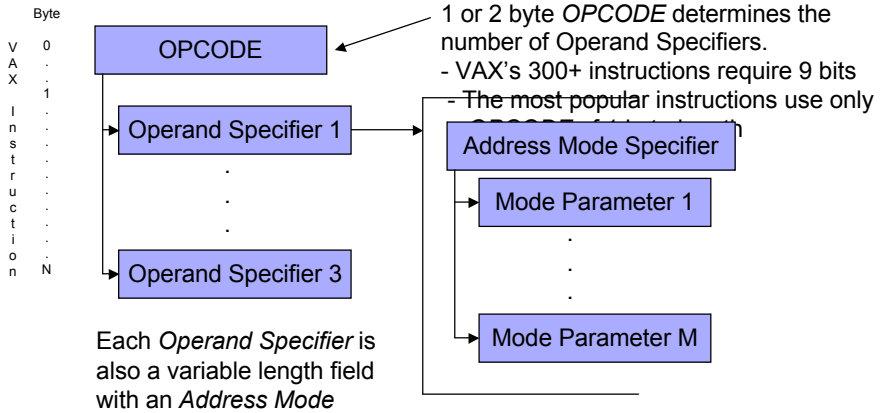
Addressing mode name	Syntax	Example	Meaning	Length of address specifier in bytes
Literal	#value	#–1	–1	1 (6-bit signed value)
Immediate	#value	#100	100	1 + length of the immediate
Register	rn	r3	r3	1
Register deferred	(rn)	(r3)	Memory[r3]	1
Byte/word/long displacement	Displacement (rn)	100(r3)	Memory[r3 + 100]	1 + length of the displacement
Byte/word/long displacement deferred	@displacement (rn)	@100(r3)	Memory[Memory [r3 + 100]]	1 + length of the displacement
Indexed (scaled)	Base mode [rx]	(r3)[r4]	Memory[r3 + r4 × *d*] (where *d* is data size in bytes)	1 + length of base addressing mode
Autoincrement	(rn)+	(r3)+	Memory[r3]; r3 = r3 + *d*	1
Autodecrement	– (rn)	–(r3)	r3 = r3 – *d*; Memory[r3]	1
Autoincrement deferred	@(rn)+	@(r3)+	Memory[Memory[r3]]; r3 = r3 + *d*	1

Figure E.2 From Appendix E

Instruction Encoding

- A one to two byte OPCODE specifies the operation, number of operands, and data type
- After the OPCODE has indicated the number of operands, each operand is represented by an Operand Specifier.
- The Operand Specifier indicates the addressing mode for the operand and the first parameter. Any further parameters must then be read in following their designated Operand Specifier.

Instruction Encoding



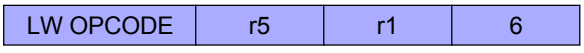
1 or 2 byte *OPCODE* determines the number of Operand Specifiers.
- VAX's 300+ instructions require 9 bits
- The most popular instructions use only

Each *Operand Specifier* is also a variable length field with an *Address Mode Specifier* followed by any of the mode specific required information.

VAX is variable length encoding

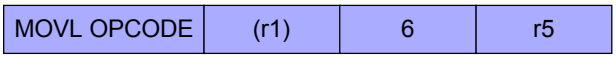
Instruction Encoding - LOAD

MIPS: lw r5, 6(r1)



6 bits 5 bits 5 bits 16 bits = 32 bits

VAX: movl 6(r1), r5



1 byte 1 byte 1 byte 1 byte = 32 bits

Call/Ret Instructions

- "calls" VAX Instruction
  - Multi-cycle instruction
  - Intended to automate and regulate the methods for preserving state before a call
  - Uses user-defined bitmask to determine which registers to save
  - Updates AP and FP to point to current frame's parameters
  - Updates PC to exec new procedure
- "ret" VAX Instruction
  - Multi-cycle instruction
  - Intended to automate and regulate the restoration of saved state after the return of a call
  - Does the opposite of the "calls" VAX Instruction

These instructions can be highly inefficient.

Code Density

```
swap(int v[], int k)
{
   int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

MIPS:
int v[] = $4
int k = $5

VAX:
int v[] = 4(ap)
int k = 8(ap)

MIPS versus VAX

Saving register	

swap: addi $29,$29, -12
sw $2, 0($29)
sw $15, 4($29)
sw $16, 8($29)
	swap: .word ^m<r0,r1,r2,r3>

Procedure body	

muli $2, $5,4
add $2, $4,$2
lw $15, 0($2)
lw $16, 4($2)
sw $16, 0($2)
sw $15, 4($2)
	movl r2, 4(a)
	movl r1, 8(a)
	movl r3, (r2)[r1]
	addl3 r0, #1,8(ap)
	movl (r2)[r1],(r2)[r0]
	movl (r2)[r0],r3

Restoring registers	

lw $2, 0($29)
lw $15, 4($29)
lw $16, 8($29)
addi $29,$29, 12	

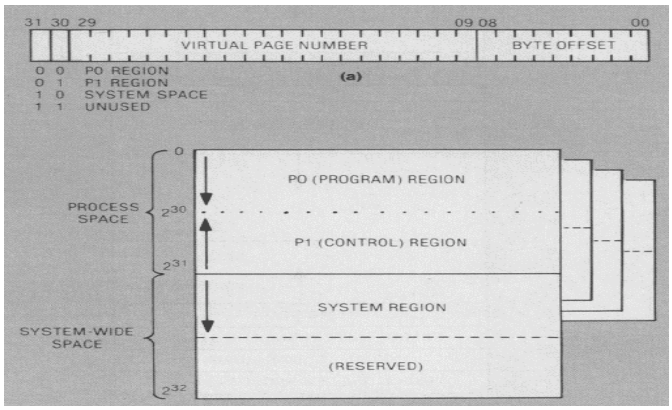
Procedure return	
jr $31	ret

FIGURE III.6 MIPS versus VAX assembly code of the procedure swap in Figure III.5 on page III-13.

Heavy code density due to CALL/RET instruction and Addressing mode

Procedure Body Memory Accesses bytes = 4 VAX: VAX 4 + 4 + 5 + 4 = 17 bytes

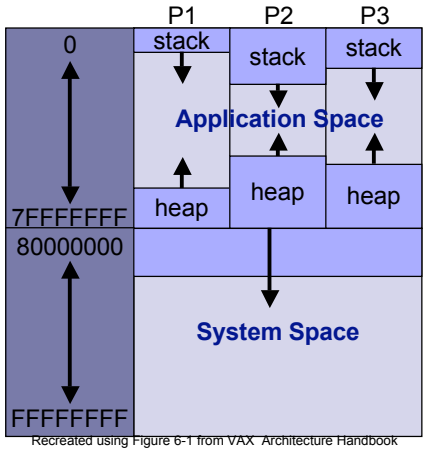
Virtual Address Layout



4GB Virtual Address Space (2GB Shared), 512Bytes Page Size

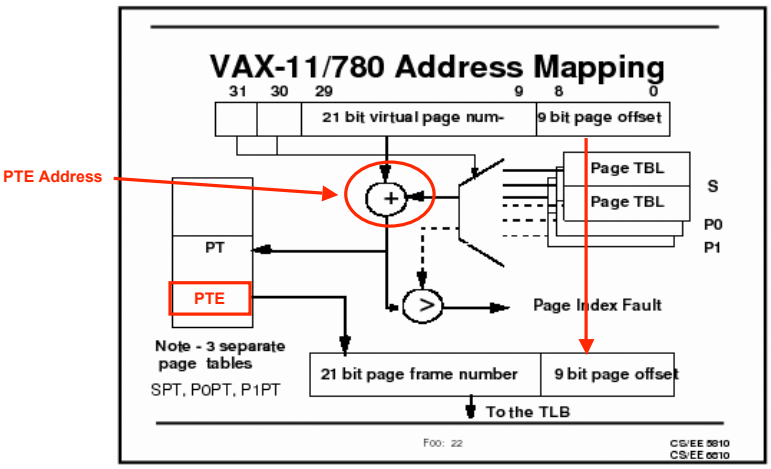
Virtual Address Extension Space

- Using mem. management with page tables, protection, and page faults VAX maps physical memory to a 32-bit (4GB) address space
- It is divided as shown in the figure to the right:



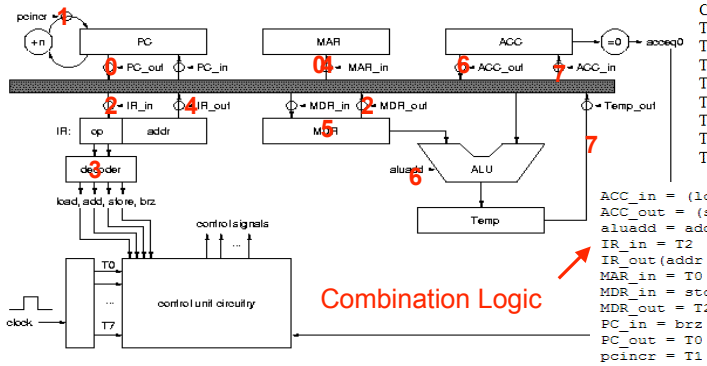
Recreated using Figure 6-1 from VAX Architecture Handbook

Virtual Address Translation



Each Region (S, P0, P1) has One-Level Page Table.
System Page Table (S) is stored in physical memory directly.
User Page Tables (P0, P1) are stored in S Region virtual address space.

Microcode - How to Control Circuit?



Control Signals Sequence – ADD
T0: PC_out, MAR_in
T1: read, pcincr
T2: MDR_out, IR_in
T3: decoding opcode in IR
T4: IR_out (addr part), MAR_in
T5: read
T6: ACC_out, aluadd
T7: TEMP_out, ACC_in, reset to T0

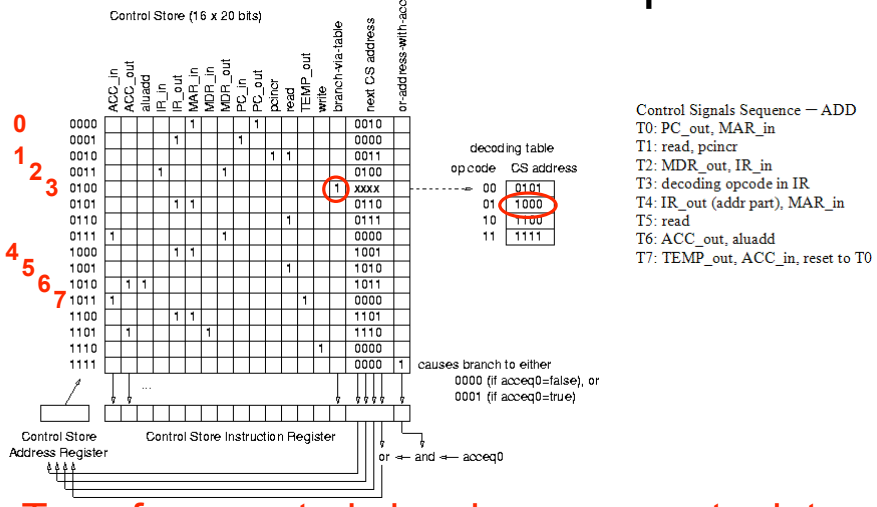
```
ACC_in = (load & T6) + (add & T7)
ACC_out = (store & T5) + (add & T6)
aluadd = add & T6
IR_in = T2
IR_out(addr part) = (load & T4) + (a
MAR_in = T0 + (load & T4) + (add & T
MDR_in = store & T5
MDR_out = T2 + (load & T6)
PC_in = brz & acceq0 & T4
PC_out = T0
pcincr = T1
read = T1 + (load & T5) + (add & T5)
TEMP_out = add & T7
write = store & T6
```

An Accumulator Example

Each instruction will be translated to a sequence of control signals.

Find a generic & simple way to control circuit?

Microcode - Concept



Control Signals Sequence – ADD
T0: PC_out, MAR_in
T1: read, pcincr
T2: MDR_out, IR_in
T3: decoding opcode in IR
T4: IR_out (addr part), MAR_in
T5: read
T6: ACC_out, aluadd
T7: TEMP_out, ACC_in, reset to T0

Transform control signals sequence to data

Horizontal & Vertical Microcode

Horizontal Microcode

Control field for each control point in the machine

μseq	μaddr	A-mux	B-mux	bus enables	register enables	…

Vertical Microcode (two-level)

Compact & simple microinstructions

Local decoded to generate all control points

ISA Instruction → Load Operand 1 … Load Operand n Opcode Operation → **Local Control Signals**

Same ISA can have different microcode designs

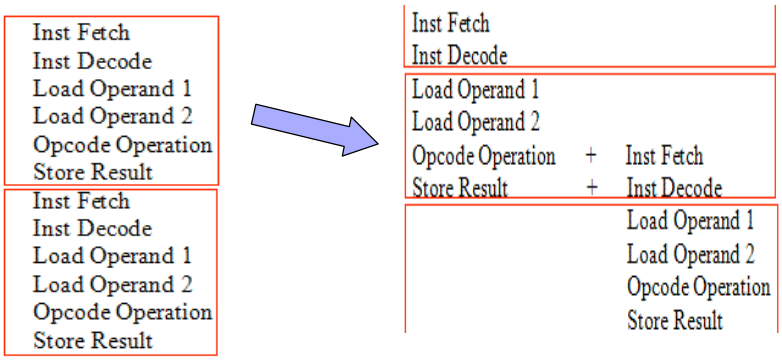
Microcode - Programming

Sample microcode for MicroVAX:

```
;     ADDx2 operation:
;
;           dst.mx <-- dst.mx + src.rx

;***  ADDx2 not RMODE ***

ADDB2..:                          ; opcode = 80
;ADDL2:                                 ; opcode = C0
;ADDW2:                                 ; opcode = A0
      ;********** Hardware dispatch **********;
      W[0]<--W[2]+W[0], SET.PSLCC, LEN(DL),     ; do the add, set psl cc's
      GOTO[WRITE.MEM..]           ; go write result to memory

;***  ADDx2 RMODE ***

ADDB2.OP..:
      ;********** Hardware dispatch **********;
      G(RN)<--G(RN)+W[0], SET.PSLCC, LEN(DL),   ; do the add, set psl cc's
      EXECUTE.IID                 ; decode next instruction
```

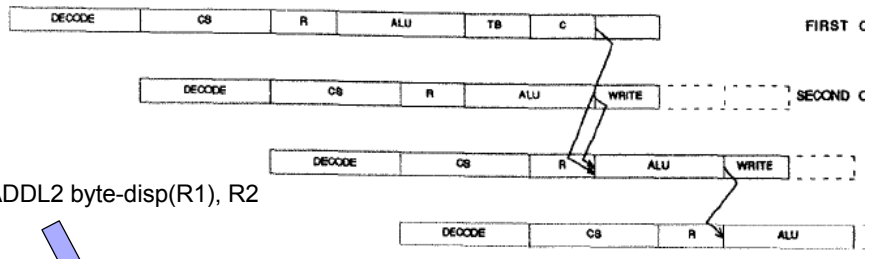
Use microcode routines to implement ISA instructions

Microprogrammed Pipeline



Microcode optimization, no hardware cost, horizontal microcode only.

Micropipeline

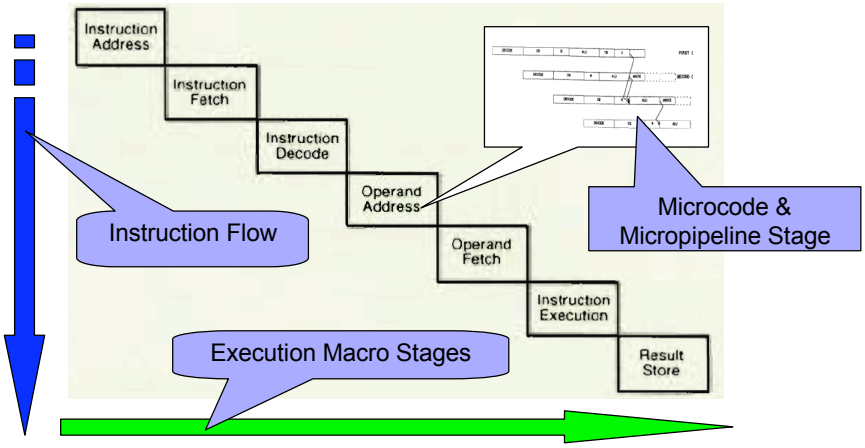


ADDL2 byte-disp(R1), R2

FIRST OPERAND: VA ← R1+disp, READ to MD0, New Decode
SECOND OPERAND: Rptr ← 2, MD1 ← R[Rptr], New Decode
ADDL2: R[Rptr] ← MD0 + MD1, New Decode
Start of next Instruction: …

Translate every CISC instruction to RISC-like microinstructions. Pipeline microinstructions like MIPS

Macropipeline



First, pipeline at VAX instruction level
Second, pipeline at microcode instruction level in some macro stages.

Summary

- VAX and its history
  ***V**irtual **A**ddress e**X**tension,* classic VAX 11-780
- VAX ISA
  32-bit Variable length CISC ISA
- VAX Virtual Address
  4GB VA, 1GB PA, 512bytes page size
- Microcode
  A generic way to control circuit
- Microcode Pipeline
  Microprogrammed pipeline, Micropipeline, Macropipeline

Thank you. ☺

Project References

Hennessy, John L., and David A. Patterson. Computer Architecture A Quantitative Approach. 3. San Francisco: Morgan Kaufmann Publishers, 2003.

VAX Architecture Handbook. Digital Equipment Corporation, 1981.