Comparative Architectures

CST Part II, 16 lectures

Lent Term 2007

David Greaves

David.Greaves@cl.cam.ac.uk

Reading material

- These slides plus the ADDITIONAL MATERIAL

- Recommended Book:
  John Hennessy & David Patterson,
  Computer Architecture: a Quantitative Approach
  (3rd ed.) 2002 Morgan Kaufmann or more recent editions

- MIT Open Courseware:
  6.823 Computer System Architecture,
  by Krste Asanovic

- The Web (see Comp Arch web page for more up-to-date links...
  `http://bwrc.eecs.berkeley.edu/CIC/`
  `http://www.chip-architect.com/`

http://www.geek.com/procspec/procspec.htm
http://www.realworldtech.com/ http://www.anandtech.com/
http://www.arstechnica.com/ http://open.specbench.org/

- comp.arch News Group

Further Reading and Reference

- M Johnson
  Superscalar microprocessor design
  1991 Prentice-Hall

- P Markstein
  IA-64 and Elementary Functions
  2000 Prentice-Hall

- A Tannenbaum,
  Structured Computer Organization (2nd ed.)
  1990 Prentice-Hall

- A Someren & C Atack,
  The ARM RISC Chip,
  1994 Addison-Wesley

- R Sites,
  Alpha Architecture Reference Manual,
  1992 Digital Press

- G Kane & J Heinrich,
  MIPS RISC Architecture
  1992 Prentice-Hall

- H Messmer,
  The Indispensable Pentium Book,
  1995 Addison-Wesley

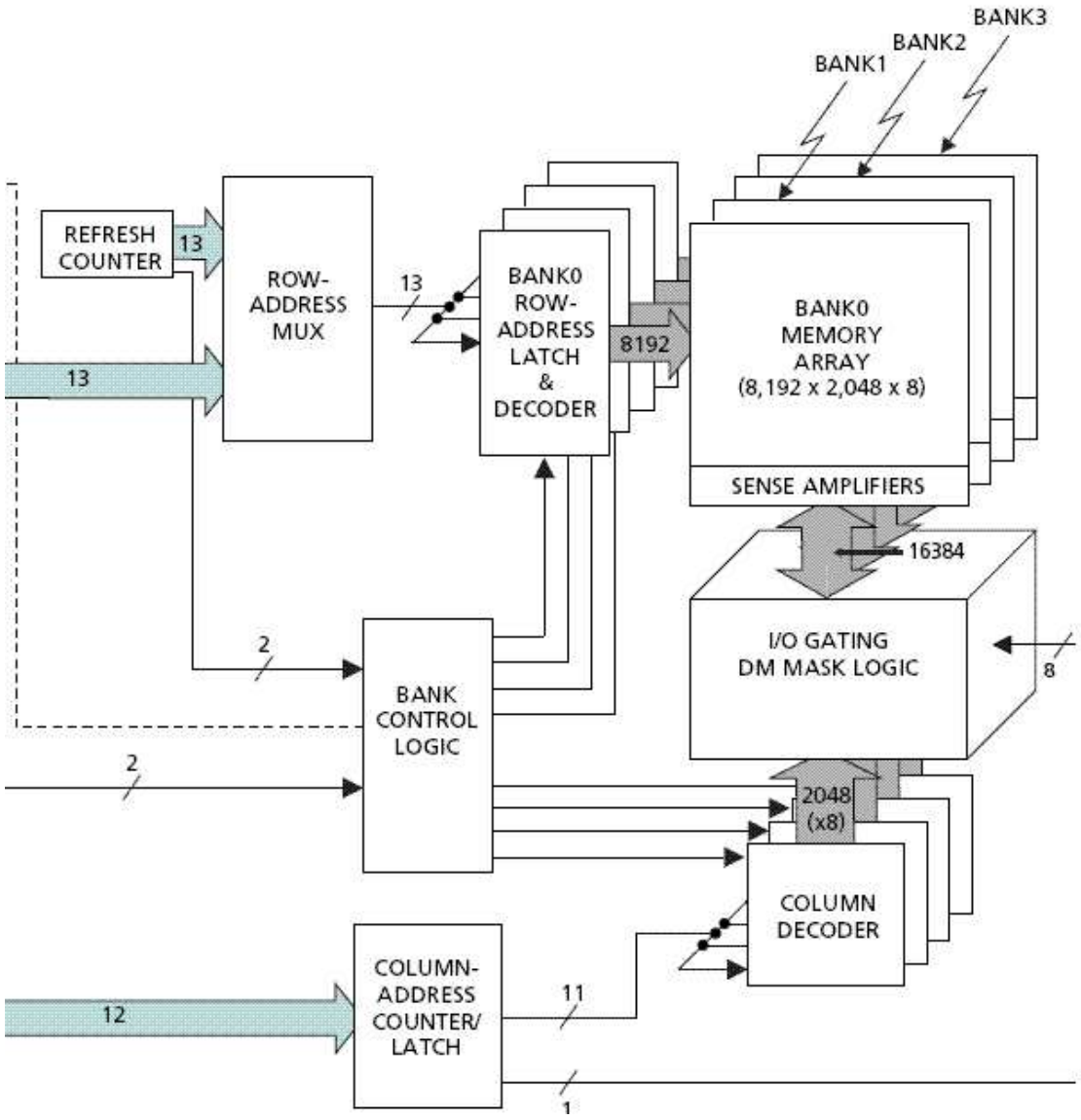
- Gerry Kane and HP,
  The PA-RISC 2.0 Architecture book,
  Prentice Hall

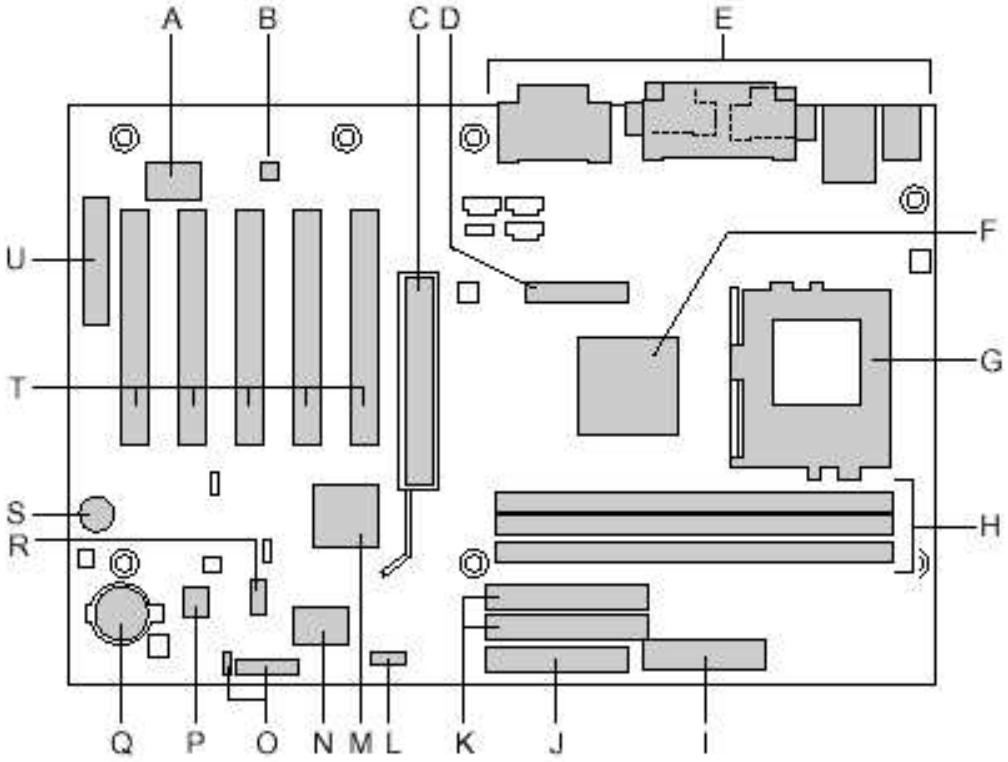
Course Pre-requisites

- Computer Design (Ib)
  - Some ARM/x86 Assembler
  - Classic RISC pipeline model
  - Load/branch delay slots
  - Cache hierarchies
  - Memory Systems
- Compilers (Ib/II)
  - Code generation
  - Linkage conventions

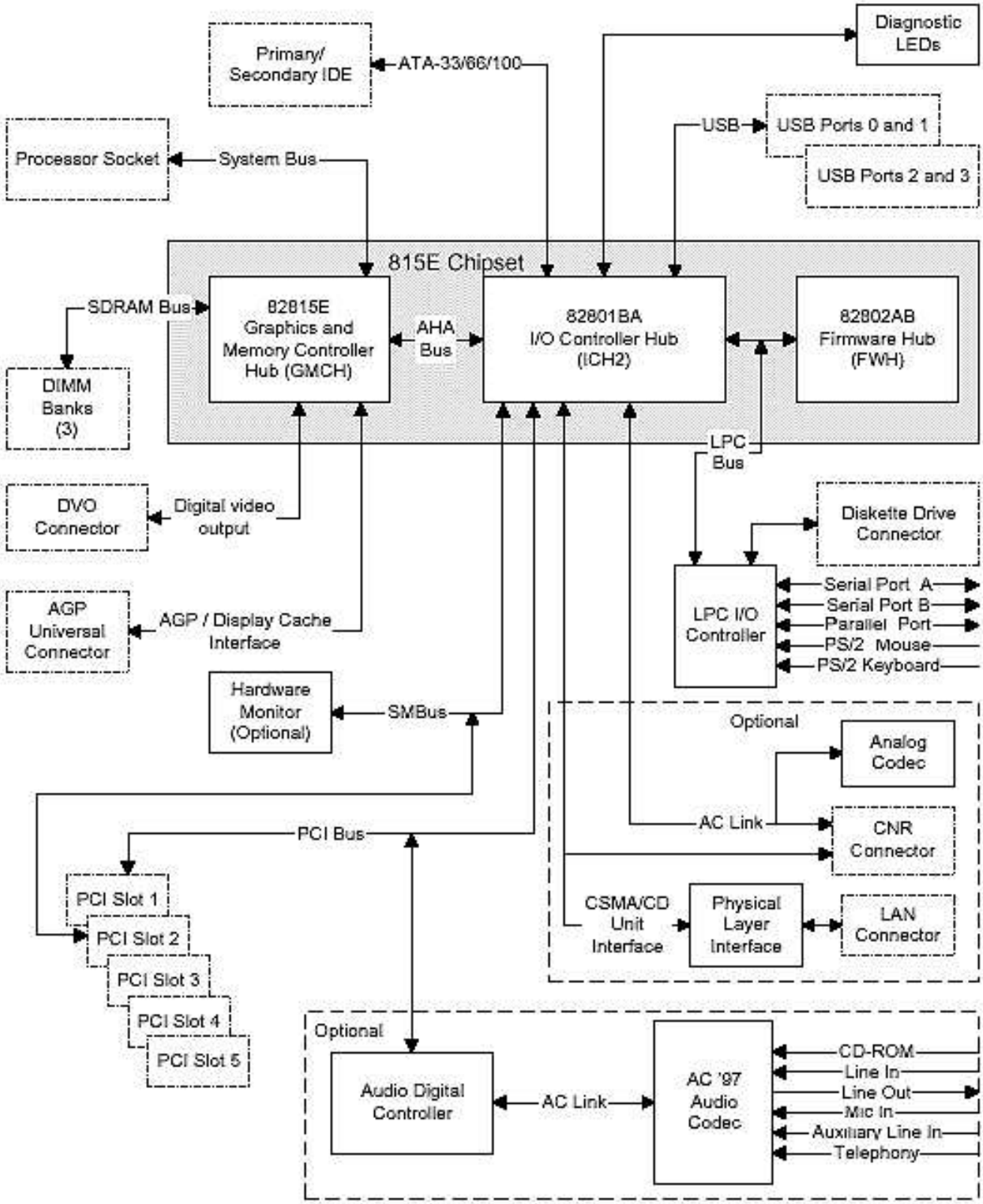
- Structured Hardware Design
  - Critical paths
  - Memories
- (Concurrent Systems)

DOUBLE DATA RATE SDRAM CHIP

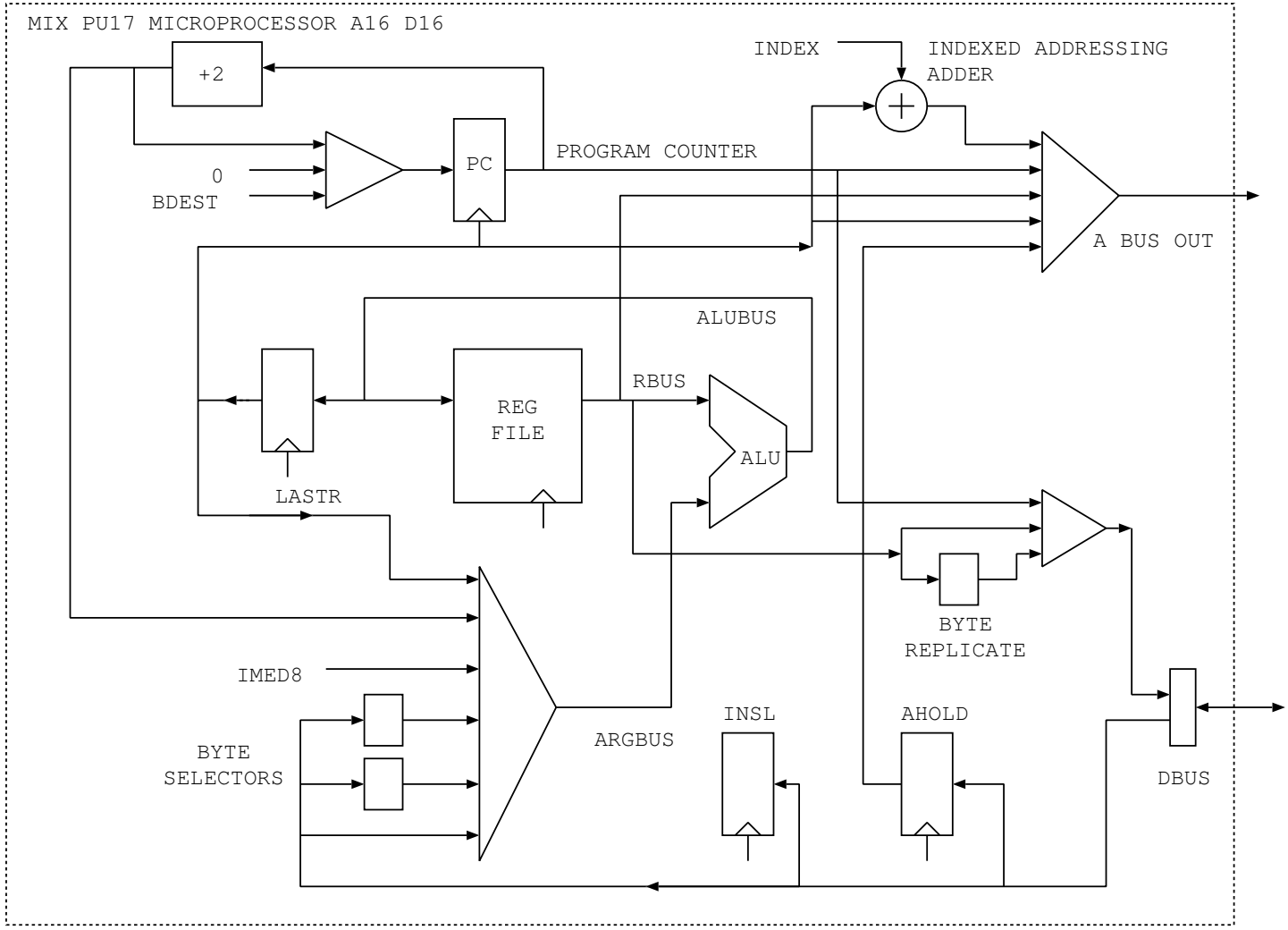




A	Creative Labs ES1373 Digital Controller (optional)	K	IDE connectors
B	AD1885 audio codec (optional)	L	Serial port B connector
C	AGP universal connector	M	Intel 82801BA I/O Controller Hub (ICH2)
D	DVO connector	N	SMSC LPC47M102 I/O Controller
E	Back panel connectors	O	Front panel connectors
F	Intel 82815E Graphics and Memory Controller Hub (GMCH)	P	Intel 82802AB 4 Mbit Firmware Hub (FWH)
		Q	Battery
		R	Front panel USB connector
G	Processor socket	S	Speaker
H	DIMM sockets	T	PCI bus add-in card connectors
I	Power connector	U	Communication and Networking Riser (CNR) connector (optional)
J	Diskette drive connector		


Diagnostic LEDs
Primary/ Secondary IDE
ATA-33/66/100
USB
USB Ports 0 and 1
USB Ports 2 and 3
Processor Socket
System Bus
815E Chipset
SDRAM Bus
82815E Graphics and Memory Controller Hub (GMCH)
AHA Bus
82801BA I/O Controller Hub (ICH2)
82802AB Firmware Hub (FWH)
DIMM Banks (3)
LPC Bus
DVO Connector
Digital video output
Diskette Drive Connector
AGP Universal Connector
AGP / Display Cache Interface
LPC I/O Controller
Serial Port A
Serial Port B
Parallel Port
PS/2 Mouse
PS/2 Keyboard
Hardware Monitor (Optional)
SMBus
Optional
Analog Codec
AC Link
CNR Connector
PCI Bus
CSMA/CD Unit Interface
Physical Layer Interface
LAN Connector
PCI Slot 1
PCI Slot 2
PCI Slot 3
PCI Slot 4
PCI Slot 5
Optional
Audio Digital Controller
AC Link
AC '97 Audio Codec
CD-ROM
Line In
Line Out
Mic In
Auxiliary Line In
Telephony

An Example Microprocessor A16 D16



```
module PU17CORE(abus16, dbus16_in, dbus16_out, clk, reset, opreq, irq, rwbar, byteop, w$
aitb);

  output [15:0] abus16;
  input [15:0] dbus16_in;
  output [15:0] dbus16_out;
  output byteop;
  input clk, reset;
  output opreq, rwbar;
  input irq;

  input waitb;                  // Acts as a clock enable essentially
                                // Wait should be changed to not gate internal cycles ?$


// Locals

  wire [15:0] pc, next_pc;
  wire [15:0] rbus, alubus, argbus;
  reg [15:0] ahold, lastr;
  wire branch_yes;                      // One if branch condition matches
  // Synchronise reset input
  reg sreset;
  always @(posedge clk) sreset <= reset;


  reg execute;                  // Execute cycle
  reg internal;                 // Internal cycle (when execute also needed)

  // Instruction decode wires
  reg update_flags;
  reg [3:0] branch_condition;
  reg regwen;
  reg [15:0] bdest;             // Branch destination
  reg [2:0] regnum;             // Register file read and write ports.

  reg write;
  reg byteop, byteopreq;
  reg imed8;
  reg argreq, argcycle;
  reg linkf;                    // Branch and link
  reg regind;                   // Register indirect
  reg idx7;                     // Even offsets to a base reg
  reg rlasave;                  // High to save PC as a return address
  reg exreq;                    // High to request an extension
  reg f0a,f0b,f0c, f1;                  // Fetch0 and fetch 1 parts of inst
  reg last_cycle;               // End cycle of current instruction
  reg f1req;                    // Request for second inst word
  reg branch;
  reg [3:0] fc;                 // ALU function code
  reg argislast;                // Used for reg to reg operations on single ported file$

  reg multiple;                 // USed for LDM/STM
  reg internal_req;
  reg [3:0] multiple_reg;       // current register to transfer in STM/LDM


  // Form a transparent latch for the old instruction.
  reg[15:0] ins_l;               // Latched instruction opcode (use in f1 onwards to re$
duce combinatorial loops in net list).
  wire [15:0] ins = (f0a) ? dbus16_in: ins_l; // Always valid.
  always @(posedge clk) if (f0a) ins_l <= dbus16_in;

wire advance = f0a | f1;
PCM pcm(pc, next_pc, advance, clk, waitb, reset, branch, bdest);
RFILE rfile(.rfile_in(alubus), .rfile_out(rbus), .regnum(regnum),
   .cen(waitb), .clk(clk), .regwen(regwen));

assign dbus16_out = (rlasave) ? pc: (byteop) ? { rbus[7:0], rbus[7:0]} :rbus;



// The ALU defaults to straight through on the b input, needing fc=12
PUALU pualu(.y(alubus), .a(rbus), .b(argbus), .fc(fc), .clk(clk), .cen(waitb),
      .update_flags(update_flags), .branch_condition(branch_condition),
      .branch_yes(branch_yes));

always @(posedge clk) if (sreset) begin
              f0a <= 0;
              f0b <= 0;
              f0c <= 0;
              f1 <= 0;
              argcycle <= 0;
              execute <= 0;
              internal <= 0;
              lastr <= 0;
              ahold <= 0;
              end



      else if (waitb) begin

      if (~execute & ~f0a & ~f1)
              begin
              f0a <= 1;       // start of day event.
              f0b <= 1;       // start of day event.
              f0c <= 1;       // start of day event.
              end
      else begin
              f0a <= last_cycle;
              f0b <= last_cycle;
              f0c <= last_cycle;
              end

      f1 <= f1req;
      argcycle <= argreq;
      byteop <= byteopreq;
      execute <= exreq;
      if (f0a | f1) ahold <= dbus16_in;

      internal <= internal_req;

      // lastr is simply the register read the cycle before.
      if (!multiple) lastr <= rbus;
      end


initial begin
      multiple = 0;
      update_flags = 0;
      branch_condition = 0;
      last_cycle = 0;
      update_flags = 0;
```

```
        rlasave = 0;

        imed8 = 0;
        write = 0;
        byteopreq = 0;
        regnum = 0;
        regwen = 0;
        argreq = 0;
        argcycle = 0;
        f1req = 0;
        fc = 4'd12;     // ALU default to load mode
        argislast = 0;
        multiple = 0;
        end

   // Instruction decoder.
   always @(ins or ins_l or f1 or f0a or f0b or f0c or execute or alubus  or branch_cond$
ition or lastr
        or multiple_reg or internal or pc or branch_yes or dbus16_in or fc) begin
        last_cycle = 0;
        fc = 4'd12;     // ALU default to load mode
        rlasave = 0;
        update_flags = 0;
        update_flags = 0;

        imed8 = 0;
        write = 0;
        regnum = 0;
        regwen = 0;
        argreq = 0;
        byteopreq = 0;
        f1req = 0;
        linkf = 0;
        idx7 = 0;
        regind = 0;
        internal_req = 0;  // not used ?
        exreq = 0;
        argislast = 0;
        branch = 0;
        bdest = 0;
        branch_condition = ins[5:2];
        multiple = 0;

        case(ins[15:12])

        4'h0, 4'h1, 4'h2, 4'h3, 4'h4, 4'h5, 4'h6, 4'h7:
        // Arith/alu immed 8 bits, one cycle.
        // If a shift, the immed arg is ignored and a shift of one is always done.
                if (f0c) begin
                        last_cycle = 1;
                        fc = ins[6:3];
                        regnum = ins[9:7];
                        regwen = (fc!=5 && fc!=13); // Not cmp or tst ;
                        update_flags = 1;
                        imed8 = 1;
                end


        4'hA,
        4'h8:   // Load from memory with index
          begin
                if (f0c) begin
                        regnum =  (ins[11:10]==3) ? 7: {1'b0, ins[11:10]};  // Read ind$
ex reg to lastr in an internal cycle
                        exreq = 1;
                        byteopreq = ins[13];
                        argreq = 1;
                        end
                if (execute) begin
                        regnum = ins_l[9:7];
                        last_cycle = 1;
                        regwen = 1;             // Indexed load with 6 bit offset
                        idx7 = 1;
                        end
          end

        4'hB,
        4'h9:   // Store to memory with index
          begin
                if (f0c) begin
                        regnum =  (ins[11:10]==3) ? 7: {1'b0, ins[11:10]}; // Read inde$
x reg to lastr in an internal cycle
                        exreq = 1;
                        byteopreq = ins[13];
                        argreq = 1;
                        end
                if (execute) begin
                        regnum = ins_l[9:7];
                        last_cycle = 1;
                        write = 1;
                        idx7 = 1;
                        end

                end

        4'hC:  // C is relative branch (BSR not supported)
                begin
                        branch_condition = ins[11:8];
                        branch = branch_yes;
                        bdest = pc + { 7 { ins[7] }, ins[7:0], 1'b0 };
                        last_cycle = 1;
                end

        4'hD:
          if (ins[11:10] == 2'b00) begin // D0 is arith reg, reg
                fc = ins[6:3];
                if (f0c) begin
                        exreq = 1;      // Read reg on first cycle
                        regnum = ins[2:0];
                        end
                if (execute) begin
                        regnum = ins_l[9:7];
                        argislast = 1;
                        last_cycle = 1;
                        regwen = (fc!=5 && fc!=13); // Not cmp or tst
                        update_flags = 1;
                        end
                end

          else if (ins[11:10] == 2'b01) begin // Load/store from memory abs 16
                regnum = ins[9:7];
                byteopreq = ins_l[6];
                if (ins[5]==0) begin // Load from an abs 16 bit address
                        if (f0b) begin
```

```
-------------------------------------------------------------------------
0-7     0xxx.   R3DEST, ALU4, IMMED8                    : Imm 8 bit
                   7,    3,  14-10, 2-0
-------------------------------------------------------------------------
8-B     10xx.    BYTEF, STOREF,   IDXR2, REG3,  IDX7,   : Indexed load/stores/add/sub
                  13,      12,   10,    7,       0    :
-------------------------------------------------------------------------
C       1100.   COND4, OFFSET8                          : Relative branches + bsr
                   8        0
-------------------------------------------------------------------------
D0      1101.00 R3DEST, ALU4, R3SRC                     : ALU reg,reg ops
                    7     3      0
-------------------------------------------------------------------------
D4      1101.01 REG3, BYTEF1,  STOREF, ABS16            : Abs16 load/store
                  7       6        5,  next
-------------------------------------------------------------------------
D8      1101.10 COND4, ABS16                            : Absolute jmp jsr
                   2,
-------------------------------------------------------------------------
DC      1101.11 RLIST8 STOREF                           : Load/store multiple
                    2,     1                            : Upwards from R7, r7 not chang$
ed
-------------------------------------------------------------------------
F0      1111.00 REG3    LinkF                           : Branch indirect
                  7,       0                            : bx, bxl
-------------------------------------------------------------------------
F4      1111.01 REG3, Immed16                           : Load immediate (mov special c$
ase)
```

```
 781                                    ; int iread(len)
 782                                    ;
 783                                    ; {
 784                                    ;
 785                                    ;   int r = 0;
 786                                    ;
 787                                    ;   int i;
 788                                    ;
 789                                    ;   for (i=0; i < len; i++)
 790                                    ;
 791   02D4 6000                           lod R0,#0   ; lti
 792   02D6 7D9C                           str  R0,[R7,#-6]   ; assign
 793                                dy29  ; anon
 794   02D8 7D8C                           lod  R0,[R7,#-6]   ; risf
 795   02DA 818C                           lod  R1,[R7,#2]   ; risf
 796   02DC 29D0                           cmp R0,R1   ; alu-l
 797   02DE 0CD84C03                       bge dy30 ; fjump F   ; cfj
 798                                    ;   {
 799                                    ;
 800                                    ; local c  [R7,#-8]
 801                                    ; s
 802   02E2 80D410DF                       lod  R1,_inpoi   ; ris
 803   02E6 61D0                           mov R0,R1   ; qasp1
 804   02E8 0900                           add R0,#1   ; qasp
 805   02EA 20D410DF                       str  R0,_inpoi   ; qasp
 806   02EE 00A4                           lodb R0,[R1]   ; risf
 807                                    ; force VR0 to 0   ; call
 808   02F0 67D1                           mov r2,r7   ; call
 809   02F2 1405                           sub r2,#12   ; call
 810   02F4 3CD88E21                       jsr _toupper   ; call
 811                                    ; force VR0 to 0   ; res
 812   02F8 7CBC                           strb R0,[R7,#-8]   ; assign
 813                                    ;     char c = toupper(*inpoi++);
 814                                    ;
 815                                    ;     while (c == ' ') c = toupper(*inpoi$
++);
 816                                    ;
 817                                dy31  ; anon
 818   02FA 7CAC                           lodb R0,[R7,#-8]   ; risf
 819   02FC 2810                           cmp R0,#32   ; alu_i
 820   02FE 04D81C03                       bne dy32 ; fjump F   ; cfj
 821   0302 80D410DF                       lod  R1,_inpoi   ; ris
 822   0306 61D0                           mov R0,R1   ; qasp1
 823   0308 0900                           add R0,#1   ; qasp
 824   030A 20D410DF                       str  R0,_inpoi   ; qasp
 825   030E 00A4                           lodb R0,[R1]   ; risf
 826                                    ; force VR0 to 0   ; call
 827   0310 67D1                           mov r2,r7   ; call
 828   0312 1605                           sub r2,#14   ; call
 829   0314 3CD88E21                       jsr _toupper   ; call
 830                                    ; force VR0 to 0   ; res
 831   0318 7CBC                           strb R0,[R7,#-8]   ; assign
 832   031A F0CA                           bra dy31   ; anon
 833                                dy32  ; anon
 834                                    ;     c = (c <= '9') ? c-'0': c-('0'+7);
 835                                    ;
 836   031C 7CAC                           lodb R0,[R7,#-8]   ; risf
 837   031E 291C                           cmp R0,#57   ; alu_i
 838   0320 10D82C03                       bgt dy33 ; fjump F   ; cfj
 839   0324 7CAC                           lodb R0,[R7,#-8]   ; risf
 840   0326 1018                           sub R0,#48   ; alu_i
 841   0328 28D83203                       bra dy34   ; anon
 842                                dy33  ; anon
 843   032C FCAC                           lodb R1,[R7,#-8]   ; risf
 844   032E 9718                           sub R1,#55   ; alu_i
 845   0330 61D0                           mov R0,R1   ; ltmv
 846                                dy34  ; anon
 847   0332 7CBC                           strb R0,[R7,#-8]   ; assign
 848                                    ;     r = (r<<4) + c;
 849                                    ;
 850   0334 7E8C                           lod  R0,[R7,#-4]   ; risf
 851   0336 4900                           asl R0,#1   ; fshif
 852   0338 4900                           asl R0,#1   ; fshif
 853   033A 4900                           asl R0,#1   ; fshif
 854   033C 4900                           asl R0,#1   ; fshif
 855   033E FCAC                           lodb R1,[R7,#-8]   ; risf
 856   0340 09D0                           add R0,R1   ; alu-l
 857   0342 7E9C                           str  R0,[R7,#-4]   ; assign
 858   0344 7D8C                           lod  R0,[R7,#-6]   ; risf
 859   0346 0900                           add R0,#1   ; qas
 860   0348 7D9C                           str  R0,[R7,#-6]   ; qasmi
 861   034A C7CA                           bra dy29   ; anon
 862                                dy30  ; anon
 863                                    ;   }
 864                                    ;
 865                                    ;   return r;
 866                                    ;
 867   034C 7EAC                           lodb R0,[R7,#-4]   ; risf
 868                                    ; force VR0 to 0   ; loadtod0
 869   034E 7F8F                           lod R6,[R7,#-2]   ; cr
 870   0350 808F                           lod r7,[r7]   ; cr
 871   0352 00F3                           ret   ; cr
 872
 873
 874                                    ; Routine mymon_dispatch
 875                                    ; forced litpool here
 876                                    .align 2
 877                  _mymon_dispatch   .global
 878   0354 809B                           str r7,[r2]
 879   0356 E2D3                           mov r7,r2
 880   0358 7F9F                           str R6,[R7,#-2]
 881   035A 019C                           str R0,[R7,#2]
 882                                    ; ---------------------------------------$
----------
 883                                    ; local argv  [R7,#2]
 884                                    ; s
 885                                    ; }
 886                                    ;
 887                                    ;
 888                                    ;
 889                                    ; int mymon_dispatch(char **argv)
 890                                    ;
 891                                    ; {
 892                                    ;
 893                                    ;
 894                                    ;
 895                                    ;   if (*argv == 0 || strlen(*argv)==0) r$
eturn;
 896                                    ;
 897   035C 018C                           lod  R0,[R7,#2]   ; risf
 898   035E 8080                           lod  R1,[R0]   ; risf
 899   0360 A800                           cmp R1,#0   ; gfv
 900   0362 00D87803                       beq dy35   ; ctj
 901   0366 818C                           lod  R1,[R7,#2]   ; risf
```

Processors

General processors

- IBM 360
- MIPS, SPARC, DLX, ARM+Thumb
- Intel x86: (8080, 8086, 80386, AMD 64)
- Intel IA 84 Itanium
- VAX and 68000
- DEC Alpha

- PA-RISC, POWER
- Transmeta Crusoe
- Sun T1 Niagra

Baby microprocessors Z80, 8080, 6502, 6800 and other A16/D8.

Specialist machines: TriMedia.

Amdahl, Blaauw and Brooks: "Architecture of the. IBM. System/360"

Instruction Set Architecture

- Processor s/w interface
- Externally visible features
  - Word size
  - Operation sets
  - Register set
  - Operand types
  - Addressing modes
  - Instruction encoding
- Introduction of new ISAs now rare

- ISAs need to last several generations of implementation

- How do you compare ISAs ?
  - yields ‘best’ implementation
    - performance, price, power
    - are other factors equal?
  - ‘aesthetic qualities’
    - ‘nicest’ for systems programmers

Instruction Set Architecture

- New implementations normally backwards compatible
  - Should execute old code correctly
  - Possibly some exceptions e.g.
    - Undocumented/unsupported features
    - Self modifying code on 68K
  - May add new features e.g. FP, divide, sqrt, SIMD, FP-SIMD
  - May change execution timings
  - $\rightarrow$ CPU specific optimization
  - Can rarely remove features

    * Unless never used
    * software emulation fast enough
  - → Layers of architectural baggage
    * (8086 16bit mode on Pentium IV)

- Architecture affects ease of utilizing new techniques e.g.
  - Pipelining
  - Super-scalar (multi-issue)

- But x86 fights real hard!
  - more T's tolerable unless on critical path

Reduced Instruction Set Computers

- RISC loosely classifies a number of Architectures first appearing in the 80's
- Not really about reducing number of instructions
- Result of quantitative analysis of the usage of existing architectures
  - Many CISC features designed to eliminate the 'semantic gap' were not used
- RISC designed to easily exploit:
  - Pipelining
    - Easier if most instructions take same amount of time

- Virtual Memory (paging)
  - Avoid tricky exceptional cases
- Caches
  - Use rest of Si area

- Widespread agreement amongst architects

Amdahl's Law

- Every 'enhancement' has a cost:
  - Would Si be better used elsewhere?
    - e.g. cache
  - Will it slow down other instructions?
    - e.g. extra gate delays on critical path
    - $\longrightarrow$ longer cycle time
- Even if it doesn't slow anything else down, what overall speedup will it give?
- size and delay

$$speedup = \frac{execution\ \ time\ \ for\ \ entire\ \ task\ \ without\ \ using\ \ enhancement}{execution\ \ time\ \ for\ \ entire\ \ task\ \ using\ \ enhancment\ \ when\ \ possible}$$

Amdahl's Law :2

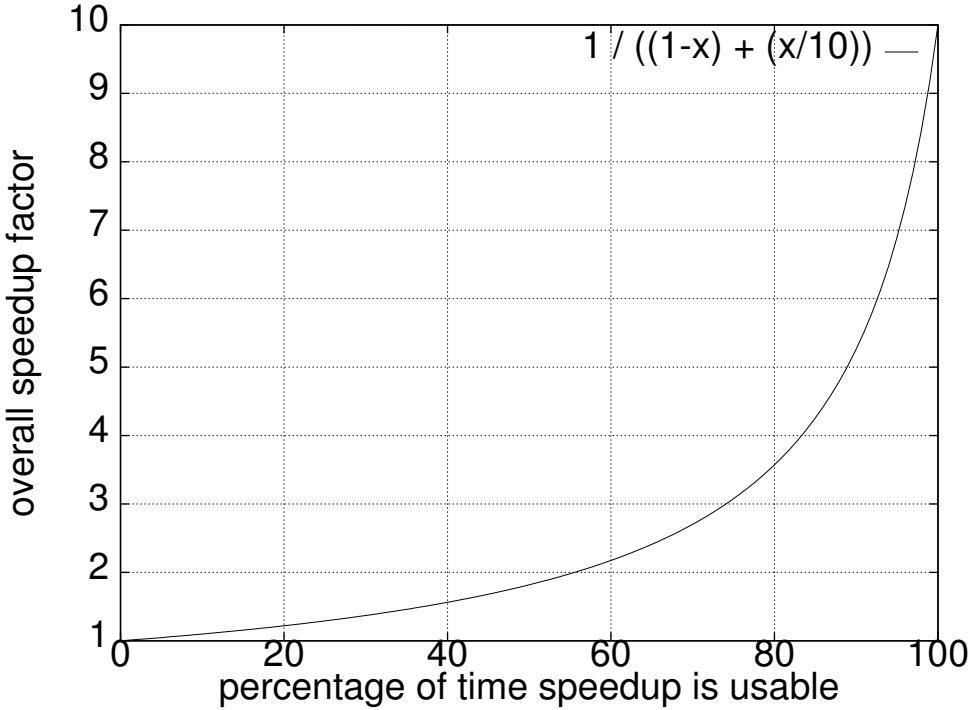
- How frequently can we use enhancement?
  - examine instruction traces e.g. SPEC
  - will code require different optimization?
  - $Fraction_{enhanced}$
- When we can use it, what speedup will it give?
  - $Speedup_{enhanced}$
  - e.g. cycles before/cycles after

$$Speedup_{overall} = \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

$\rightarrow$ **Spend resources where time is spent**

Optimize for the common case

Amdahl's Law for Speedup=10



Amdahl's Law Example

- FPSQRT is responsible for 20% of execution time in a (fictitious) critical benchmark
- FP operations account for 50% of execution time in total
- Proposal A:
  - New FPSQRT hardware with 10x performance

$$speedup_A = \frac{1}{(1 - 0.2) + \frac{0.2}{10}} = \frac{1}{0.82} = 1.22$$

- Proposal B:
  - Use Si area to double speed all FP operations

$$speedup_B = \frac{1}{(1 - 0.5) + \frac{0.5}{2}} = \frac{1}{0.75} = 1.33$$

- $\rightarrow$ Proposal B is better

- (Probably much better for other users)

Word Size

- Native size of an integer register
  - 32bits on ARM, MIPS II, x86 32bit mode
  - 64bits on Alpha, MIPS III, SPARC v8, PA-RISC v2
- NOT size of FP or SIMD registers
  - 64 / 128 bit on Pentium III
- NOT internal data-path width
  - 64bit internal paths in Pentium III
- NOT external data-bus width

- 8bit Motorola 68008
- 128bit Alpha 21164

- NOT size of an instruction
  - Alpha, MIPS, etc instructions 32bit

- But, 'word' also used as a type size
  - 4 bytes on ARM, MIPS
  - 2 bytes on Alpha, x86
    - longword = 4 bytes, quadword = 8

64bit *vs* 32bit words

- Alpha, MIPS III, SPARC v8, PA-RISC v2

✔ Access to a large region of address space from a single pointer

  – large data-structures

  – memory mapped files

  – persistent objects

✔ Overflow rarely a concern

  – require fewer instructions

✘ Can double a program's data size

- need bigger caches, more memory b/w

✘ May slow the CPU's max clock speed

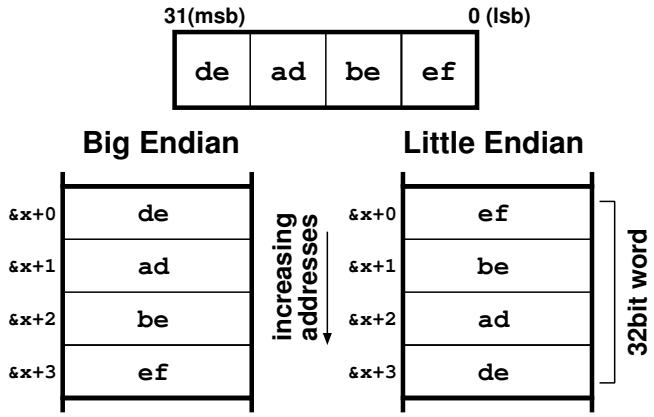
- Some programs gain considerably from 64bit, others get no benefit.

- Some OS's and compilers provide support for 32bit binaries

Byte Sex

- Little Endian camp
  - Intel, Digital
- Big Endian camp
  - Motorola, HP, IBM
  - Sun: 'Network Endian', JAVA
- Bi-Endian Processors
  - Fixed by motherboard design
  - MIPS, ARM
- Endian swapping instructions

```
int   x= 0xdeadbeef;
char *p= (char*)&x;
if(*p == 0xde) printf("Big Endian");
if(*p == 0xef) printf("Little Ebdian");
```

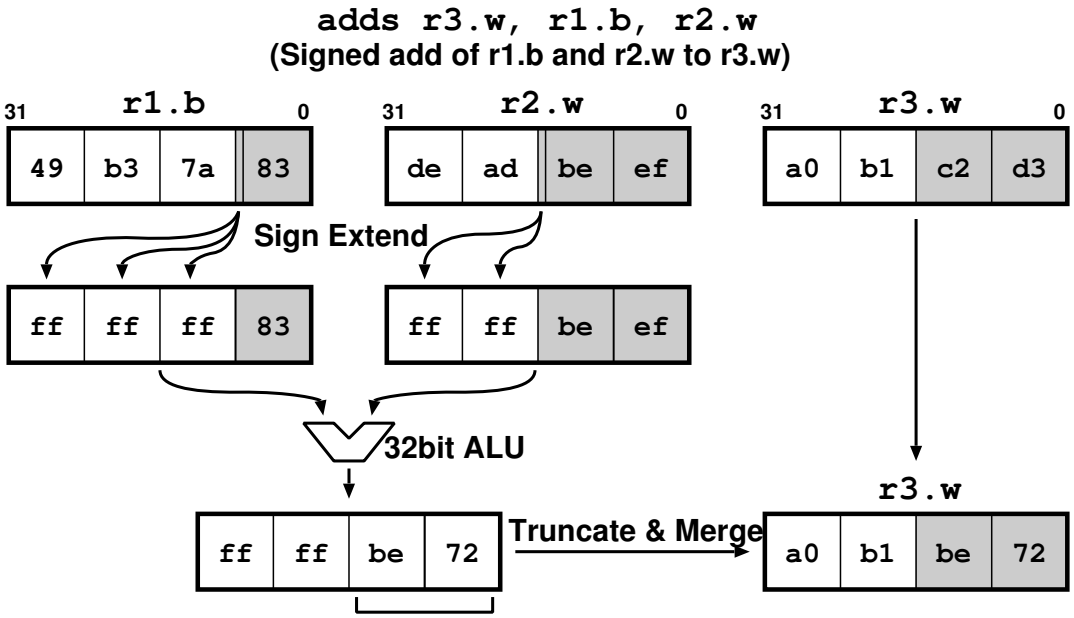


Data Processing Instructions

- 2's Complement Arithmetic
  - add, subtract, multiply, compare, multiply
  - some: divide, modulus
- Logical
  - and, or, not, xor, bic, . . .
- Shift
  - shift left, logical shift right, arithmetic shift right
  - some: rotate left, rotate right

Operand Size

- CISC
  - 8,16,32 bit operations
  - zero/sign extend sources
    - need unsigned/signed instrs
  - merge result into destination
  - some even allow mixed size operands



- RISC
  - Word size operations only
  - (except 64bit CPUs often support 32bit ops)
  - Pad `char` and `short` to word

(Zero/Sign Extension)

- Unsigned values: zero extend
  - e.g. 8bit values to 32bit values

```
unsigned char a; int b;
 and  b ← a, #0xff
```

- Signed values: sign extend
  - e.g. 8bit values to 32bit values
  - Replicate sign bit

```
char a; int b;
 lsl  b ← a, #24
 asr  b ← b, #24
```

- C: 32bit to 8bit

– Just truncate

```
and  b ← a, #0xff
```

CISC instructions RISC dropped

- Emulated in RISC:

```
move   r1 ← r2   e.g. or   r1 ← r2, r2
zero   r1        e.g. xor  r1 ← r1, r1
neg    r1        e.g. sub  r1 ← #0, r1
nop              e.g. or   r1 ← r1, r1
sextb  r1 ← r2   e.g. lsl  r1 ← r2, #24;
                      asr  r1 ← r1, #24
```

- Used too infrequently:
  - POLY, polynomial evaluation (VAX)
  - BCD, bit-field operations (68k)
  - Loop and Procedure call primitives
    - Not quite right for every HLL

    * Unable to take advantage of compiler's analysis

- Exceptions & interrupts are awkward:
  - memcpy/strcmp instructions

New Instructions

- integer divide, sqrt
- popcount, priority_encode
- Integer SIMD (multimedia)
  - Intel MMX, SPARC VIS, Alpha, PA-RISC MAX
  - MPEG, JPEG, polygon rendering
  - parallel processing of packed sub-words
  - E.g. 8x8, 4x16 bit packed values in 64b word
  - arithmetic ops with 'saturation'
    - s8 case: 125+4 = 127

  - min/max, logical, shift, permute
  - RMS error estimation (MPEG encode)
  - Will compilers ever use these instrs?

- FP SIMD (3D geometry processing)
  - E.g. 4x32 bit single precision
  - streaming vector processing
  - Intel SSE, AMD 3D-Now, PPC AltiVec

- prefetch / cache hints (e.g. non-temporal)

- Maintaining backwards compatiblity
  - Use alternate routines

– Query CPU feature set

Registers and Memory

- Register set types
  - Accumulator architectures
  - Stack
  - GPR
- Number of operands
  - 2
  - 3
- Memory accesses
  - any operand

– one operand

– load-store only

Accumulator Architectures

- Register implicitly specified
- E.g. 6502, 8086 (older machines)

```
LoadA   foo
AddA    bar
StoreA  res
```

- Compact instruction encoding
- Few registers, typically ≤ 4 capable of being operands in arithmetic operations
- Forced to use memory to store intermediate values

- Registers have special functions
  - e.g. loop iterators, stack pointers
- Compiler writers don't like non-orthogonality

Stack Architectures

- Operates on top two stack items
- E.g. Transputer, (Java)

```
Push  foo
Push  bar
Add
Pop   res
```

- Stack used to store intermediate values
- Compact instruction encoding
- Smaller executable binaries, good if:
  - memory is expensive

  - downloaded over slow network

- Fitted well with early compiler designs

General Purpose Register Sets

- Post 1980 architectures, both RISC and CISC
- 16,32,*128* registers for intermediate values
- Separate INT and FP register sets
  - Int ops on FP values meaningless
  - RISC: Locate FP regs in FP unit
- Separate Address/Data registers
  - address regs used as bases for mem refs
  - e.g. Motorola 68k

– not favoured by compiler writers ($8 + 8 \neq 16$)

– RISC: Combined GPR sets

Load-Store Architecture

- Only load/store instructions ref memory

- The RISC approach

→ Makes pipelining more straightforward

```
Load   r1 ← foo
Load   r2 ← bar
Add    r3 ← r1, r2
Store  res← r3
```

- Fixed instruction length (32bits)

- 3 register operands

- Exception: ARM-Thumb, MIPS-16 is two operand
  - more compact encoding (16bits)

Register-Memory

- ALU instructions can access 1 or more memory locations

- E.g. Intel x86 32bit modes
  - 2 operands
  - can't both be memory

```
Load   r1←foo
Add    r1←bar
Store  res←r1
```

- E.g. DEC VAX
  - 2 and 3 operand formats
  - fully orthogonal

```
Add  res←bar,foo
```

- Fewer instructions
  - Fewer load/stores
  - Each instruction may take longer
  - $\rightarrow$ Increased cycle time
- Variable length encoding
  - May be more compact
  - May be slower to decode

Special Registers : 1

- Zero register
  - Read as Zero, Writes discarded
  - e.g. Alpha, Mips, Sparc, IA-64
  - Data move: `add r2 ← r1, r31`
  - nop: `add r31 ← r31, r31`
  - prefetch: `ldl r31 ← (r1)`
  - Zero is a frequently used constant
  - IBM 360: register zero reads zero as an index register
- Program Counter
  - NOT usually a GPR
  - Usually accessed by special instructions e.g. branch, branch and link, jump
  - But, PC is GPR r15 on ARM

Special Registers : 2

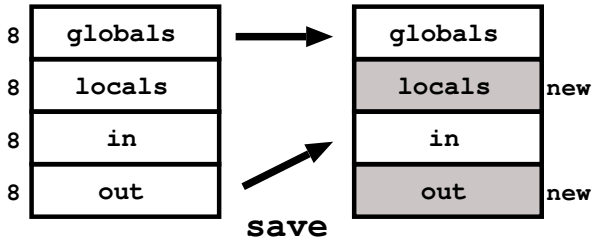
- Condition code (Flag) registers
  - Carry, Zero, Negative, Overflow
  - Used by branches, conditional moves
  - Critical for pipelining and super-scalar
  - CISC: one CC reg updated by all instructions
  - ARM, SPARC: one CC reg, optionally updated
  - PowerPC: multiple CC regs (instr chooses)
  - IA64: 64 one bit predicate regs
  - Alpha, MIPS: no special CC regs
- Link registers

- Subroutine call return address
- CISC: pushed to stack
- RISC: saved to register
  - register conventions
  - only push to stack if necessary
- Jump target/link regs (PowerPC, IA-64)
- fixed GPR (r14, ARM) (r31,MIPS)
- GPR nominated by individual branch (Alpha, IBM 360)

Register Conventions

- Linkage (Procedure Call) Conventions
  - Globals: `sp, gp` etc.
  - Args: First (4-6) args (rest on stack)
  - Return value: (1-2)
  - Temps: (8-12)
  - Saved: (8-9) Callee saves
- Goal: spill as few registers as possible in total
- Register Windows (SPARC)
  - `save` and `restore`

- 2-32 sets of windows in ring
- 16 unique registers per window
- spill/fill windows to special stack



- IA-64: Allows variable size frames
  - 32 globals
  - 0-8 args/return, 0-96 locals/out args
  - h/w register stack engine operates in background

Classic RISC Addressing Modes

- Register
  - `Mov r0 ← r1`
  - `Regs[r0] = Regs[r1]`
  - Used when value held in register
- Immediate
  - `Mov r0 ← 42`
  - `Regs[r0] = 42`
  - Constant value limitations
- Register Indirect

- `Ldl r0 ← [r1]`
- `Regs[r0] = Mem[ Regs[r1] ]`
- Accessing variable via a pointer held in reg

- Register Indirect with Displacement
  - `Ldl r0 ← [r1, #128]`
  - `Ldl r0 ← 128(r1)`
  - `Regs[r0] = Mem[ 128 + Regs[r1] ]`
  - Accessing local variables

Less RISCy addr modes

- ARM and PowerPC

- Register plus Register (Indexed)
  - `Ldl r0 ← [r1,r2]`
  - `Regs[r0] = Mem[ Regs[r1] + Regs[r2] ]`
  - Random access to arrays
  - e.g. r1=base, r2=index

- Register plus Scaled Register
  - `Ldl r0 ← [r1, r2, asl #4]`
  - `Regs[r0] = Mem[ Regs[r1] + (Regs[r2]≪4)]`

  - Array indexing
  - sizeof(element) is power of 2, r2 is loop index

- Register Indirect with Displacement and Update
  - Pre inc/dec `Ldl r0` $\leftarrow$ `[r1!, #4]`
  - Post inc/dec `Ldl r0` $\leftarrow$ `[r1], #4`
  - C `*(++p)` and `*(p++)`
  - Creating stack (local) variables
  - Displacement with post update is IA-64's only addressing mode

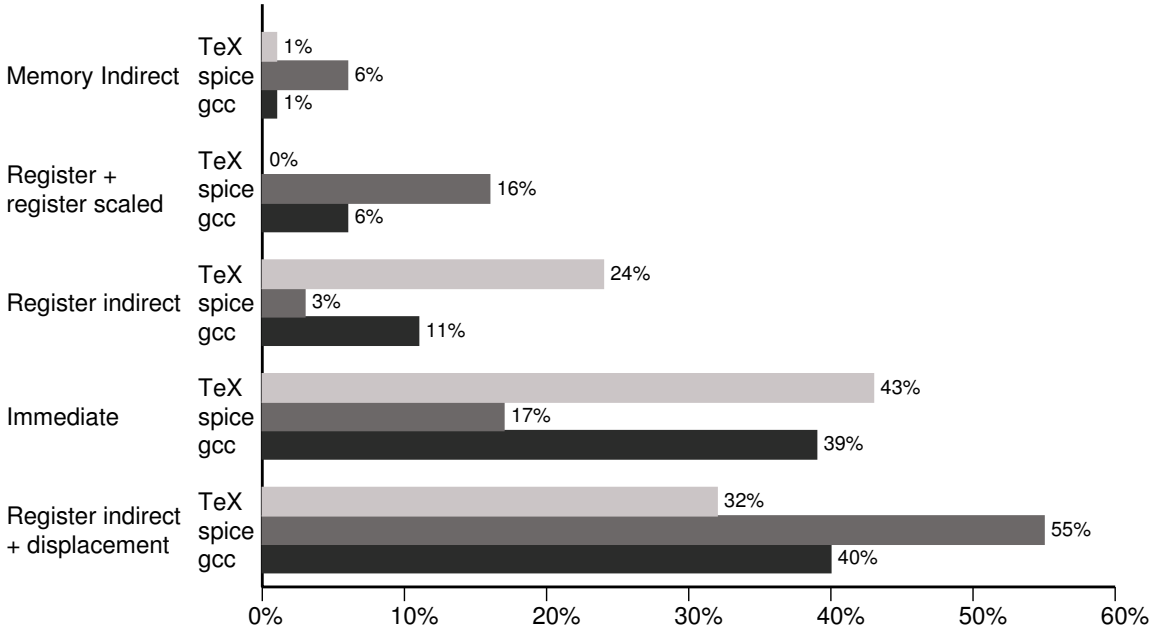
CISC Addressing Modes

- Direct (Absolute)
  - `Mov r0 ← (1000)`
  - `Regs[r0] = Mem[1000]`
  - Offset often large
  - x86 Implicit base address
  - Most CISCs
- Memory Indirect
  - `Mov r0 ← @[r1]`
  - `Regs[r0] = Mem[ Mem[ Regs[r1] ] ]`

  - Two memory references,
  - C `**ptr`, linked lists

- PC Indirect with Displacement
  - `Mov r0 ← [PC, #128]`
  - `Regs[r0] = Mem[ PC + 128 ]`
  - Accessing constants

Why did RISC choose these addressing modes?



- RISC
  - immediate
  - register indirect with displacement

- ARM, PowerPC reduce instruction counts by adding:
  - register + register scaled
  - index update

Immediates and Displacements

- CISC: As instructions are variable length, immediates and displacements can be any size (8,16,32 bits)

- RISC: How many spare bits in instruction format?

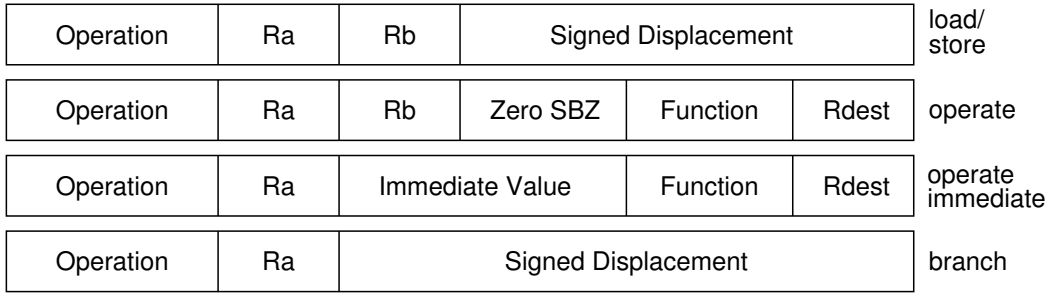
- Immediates
  - used by data-processing instructions
  - usually zero extended (unsigned)
    - `add` $\longrightarrow$ `sub`
    - `and` $\longrightarrow$ `bic`
  - For traces on previous slide:
    50-70% fit in 8bits, 75-80% in 16bits

- IA-64 22/14, MIPS 16, Alpha 8,
  ARM 8 w/ shift

- Displacement values in load and stores
  - Determine how big a data segment you can address without reloading base register
  - usually sign extended
  - MIPS 16, Alpha 16, ARM 12, IA-64 9, IBM-360 12

Instruction Encoding

RISC: small number of fixed encodings of same length

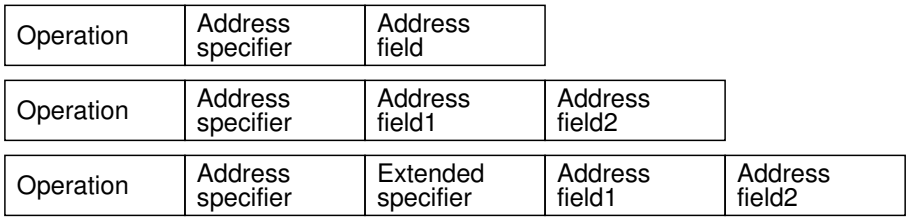


RISC instruction words are 32 bit

IA-64 packs three 41 bit instructions into a 128 bit 'bundle'

VAX: fully variable. Operands specified independently

x86: knows what to expect after first couple of bytes



Code Density Straw Poll

- CISC: Motorola 68k, Intel x86
- RISC: Alpha, Mips. PA-RISC
- Very rough-figures for 68k and Mips include statically linked libc

arch	text	data	bss	total	filename
x86	29016	14861	468	44345	gcc
68k	36152	4256	360	40768	
alpha	46224	24160	472	70856	
mips	57344	20480	880	78704	
hp700	66061	15708	852	82621	
x86	995984	156554	73024	1225562	gcc-cc1
alpha	1447552	272024	90432	1810008	
hp700	1393378	21188	72868	1487434	
68k	932208	16992	57328	1006528	
mips	2207744	221184	76768	2505696	
68k	149800	8248	229504	387552	pgp
x86	163840	8192	227472	399504	
hp700	188013	15320	228676	432009	
mips	188416	40960	230144	459520	
alpha	253952	57344	222240	533536	

- CISC text generally more compact,
  but not by a huge amount

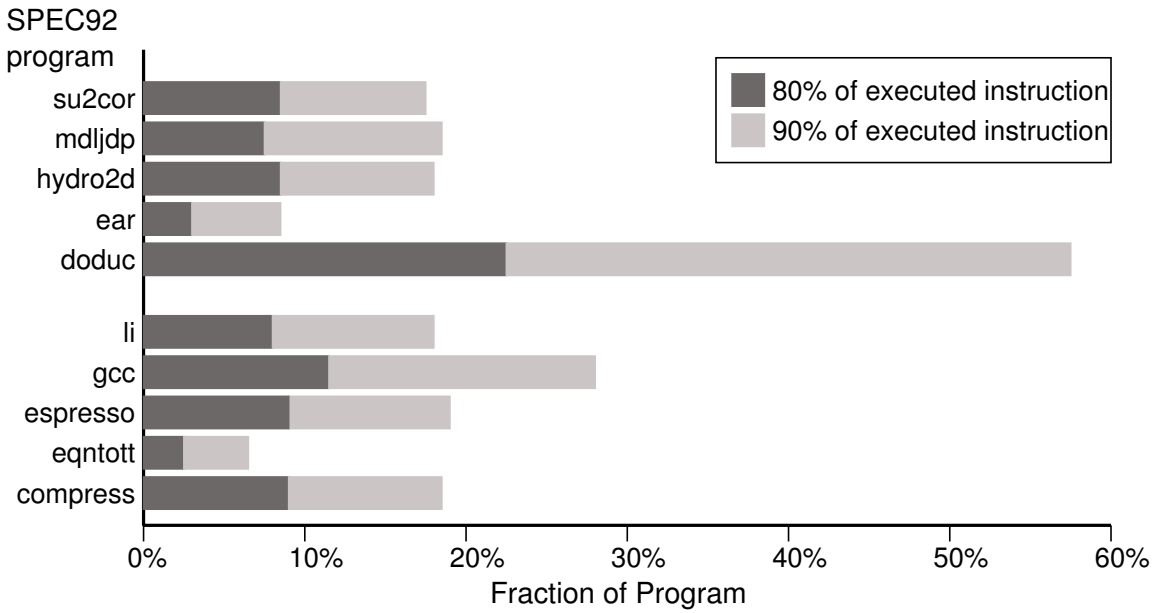
- Alpha's 64bit data/bss is larger

Code Density

- Important if:
  - Memory is expensive
    - * can be in embedded applications
    - * eg. mobile phones
    - $\Rightarrow$ ARM Thumb, MIPS-16
  - Executable loaded over slow network
    - * Though Java not particularly dense!
- Speed vs. size optimization tradeoffs
  - loop unrolling
  - function inlining

– branch/jump target alignment

Instruction caches



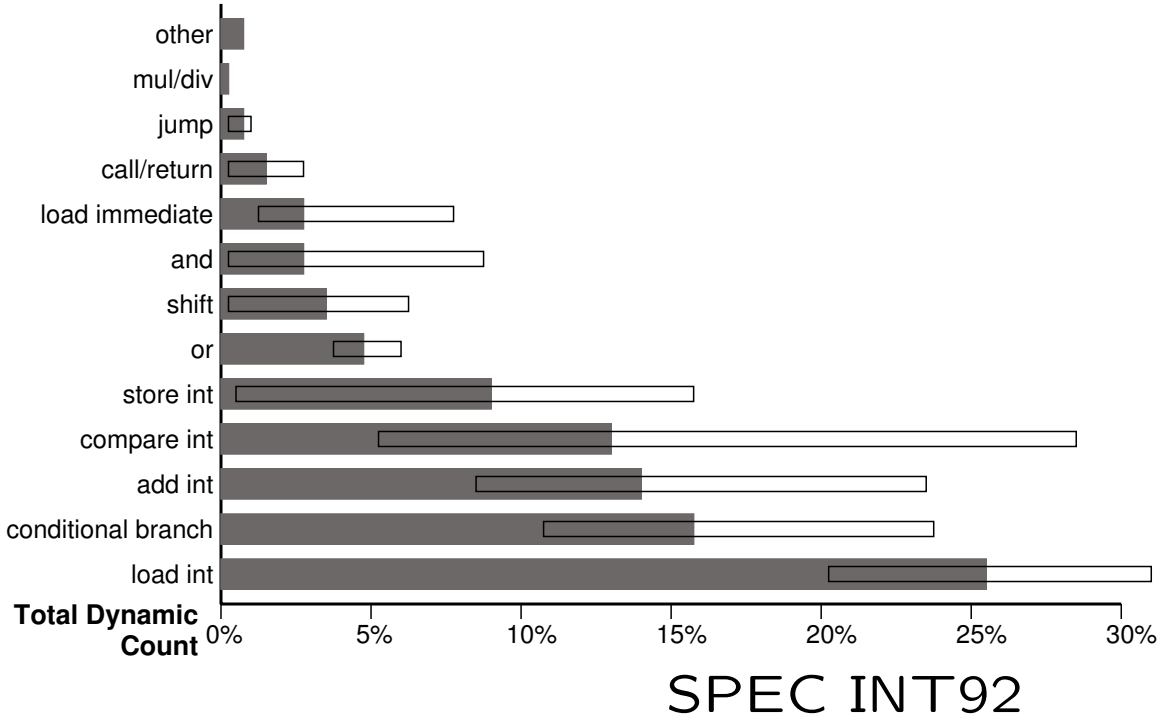
Fraction of program responsible for 80% and 90% of instruction executions

- Caches generally solve I-stream b/w requirements
  - 4bytes x 1GHz x 2-4 instrs = 8-16GB/s !
  - Loops are common! (90% in 10%)
  - Internal I-caches often get 95%+ hit-rates

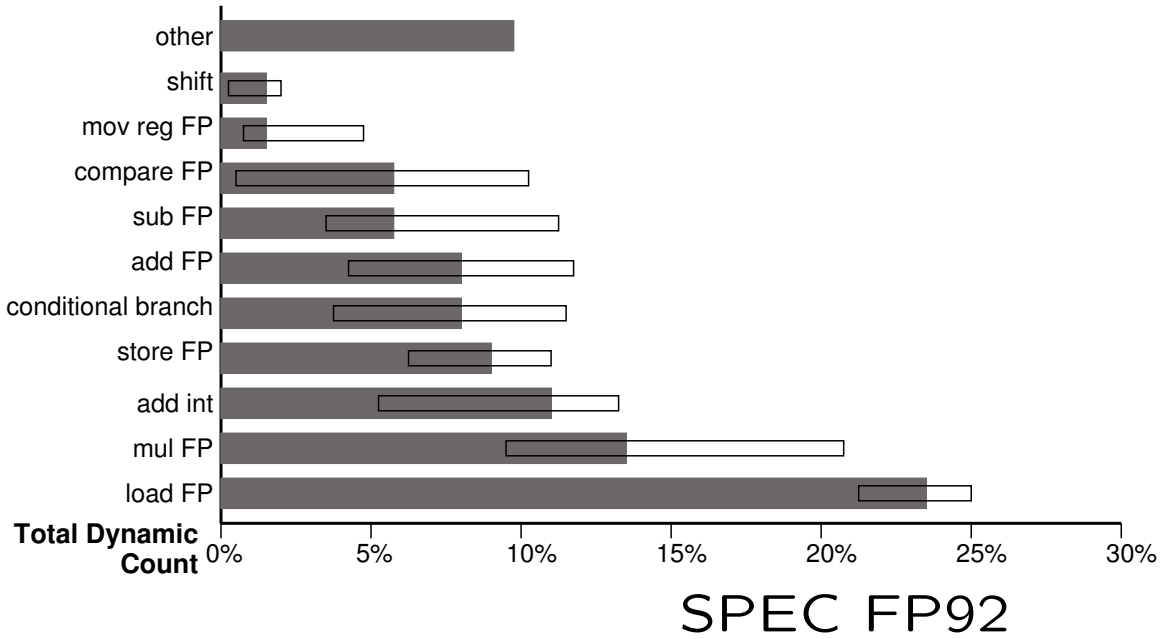
  - Code density not usually a performance issue
    * assuming decent compilers and app design
    * code out-lining (trace straightening) vs. function in-lining and loop unrolling

- D-Cache generally much more of a problem

Instruction Mix



Instruction mix for SPEC INT92



Instruction mix for SPEC FP92

There are no 'typical' programs

Microcoded Processors

- EDSAC and many CISC machines were microcoded.
- Op-code mapped to a start address in microcode control ROM.
- A *micro-sequencer* executes the microcode.
- Multi-cycle instructions, MUL, DIV, string operations, ...
- Useful for VM table walking.
- Malformed microcode can cause hardware bus fight!
- Horizontal and Vertical forms.
- Simple ISA instructions not microcoded but execute directly.

Horizontal microcode has a wide instruction word and no or fast decoder.

Vertical microcode is more compact but places more logic on the critical path.

Alpha had programmable microcode: procedure entry, multi-media, ...

POWER MORE IMPORTANT THAN PERFORMANCE ?

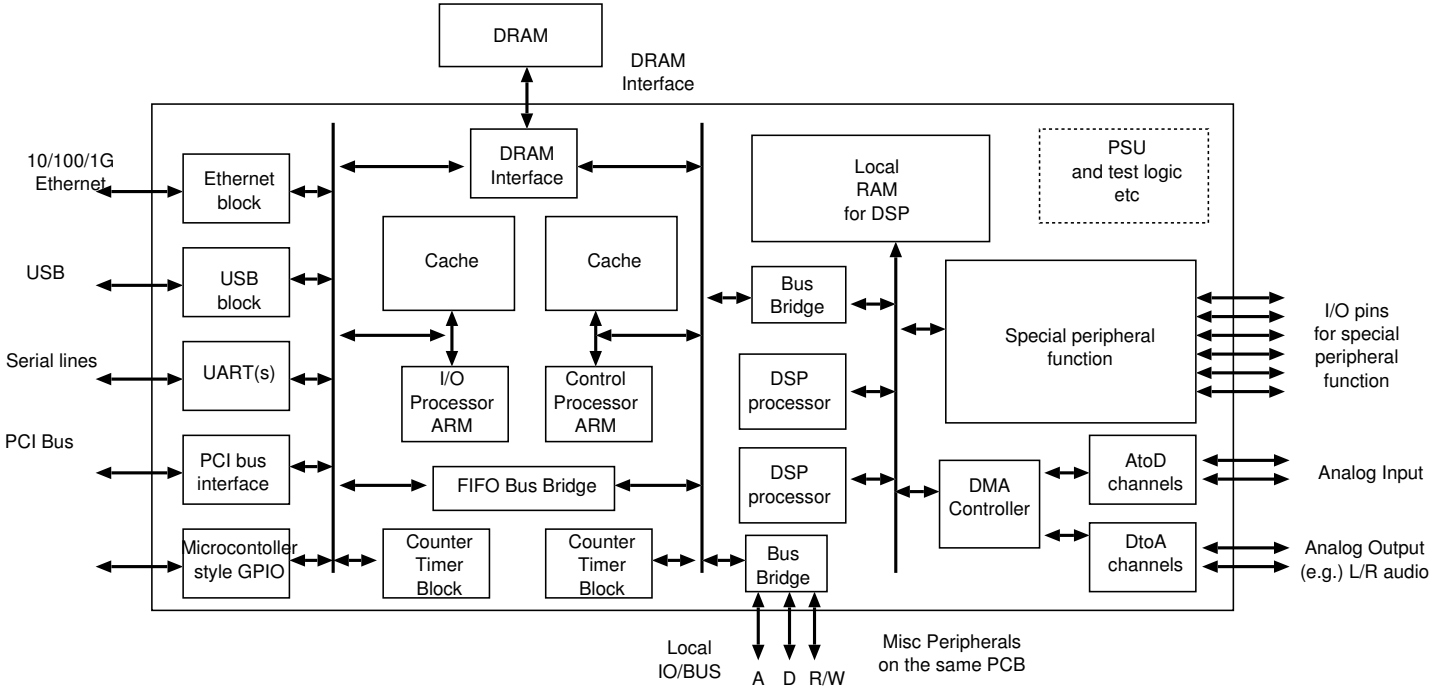
1. Battery operated PICOs
    - Intel Centrino
    - Transmeta Crusoe
    - ARM
    - Tensilica
2. Processors Everywhere
    - We own 100 computers each!
    - Maybe 10,000 by 2012

3. Joule is the unit of energy

   - One instruction on Intel XScale takes 1 nJ
   - 720 Joules/gram for Li-Fe batteries.
   - Reducing switching voltage - great power savings
   - Reducing clock frequency - only saves wasted clock cycles
   - Dynamic clock and voltage adjustment versus parallelism

From Asanovic/Devadas

1998: A Platform Chip: D32/A32 twice!



System on a Chip = SoC design.

Our platform chip has two ARM processors and two DSP processors. Each ARM has a local cache and both store their programs and data in the same offchip DRAM.

The left-hand-side ARM is used as an I/O processor and so is connected to a variety of standard peripherals. In any typical application, many of the peripherals will be unused and so held in a power down mode.

The right-hand-side ARM is used as the system controller. It can access all of the chip's resources over various bus bridges. It can access off-chip devices, such as an LCD display or keyboard via a general purpose A/D local bus.

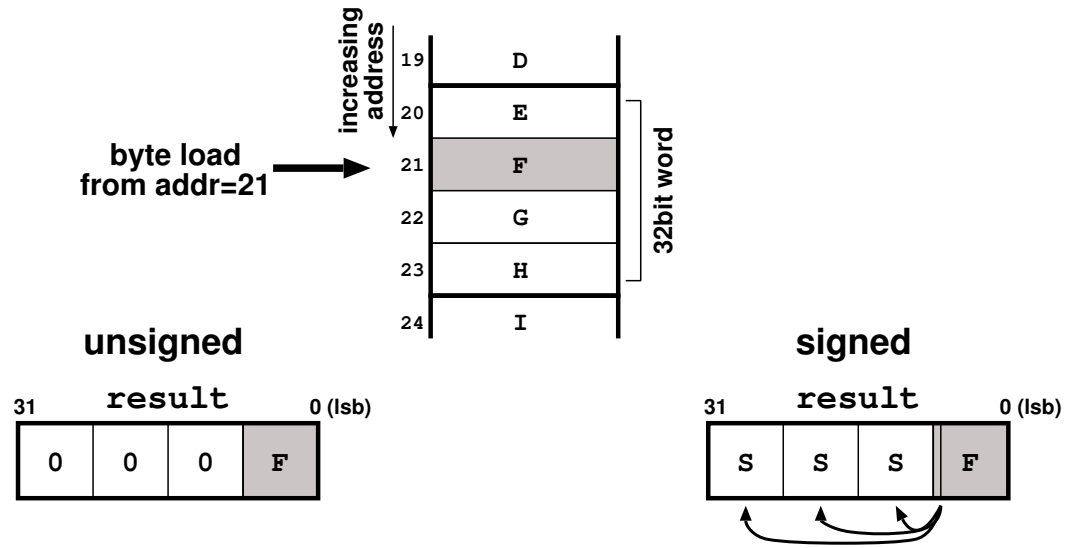
The bus bridges map part of one processor's memory map into that of another so that cycles can be executed in the other's space, allbeit with some delay and loss of performance. A FIFO bus bridge contains its own transaction queue of read or write operations awaiting completion.

The twin DSP devices run completely out of on-chip SRAM. Such SRAM may dominate the die area of the chip. If both are fetching instructions from the same port of the same RAM, then they had better be executing the same program in lock-step or else have some own local cache to avoid huge loss of performance in bus contention.

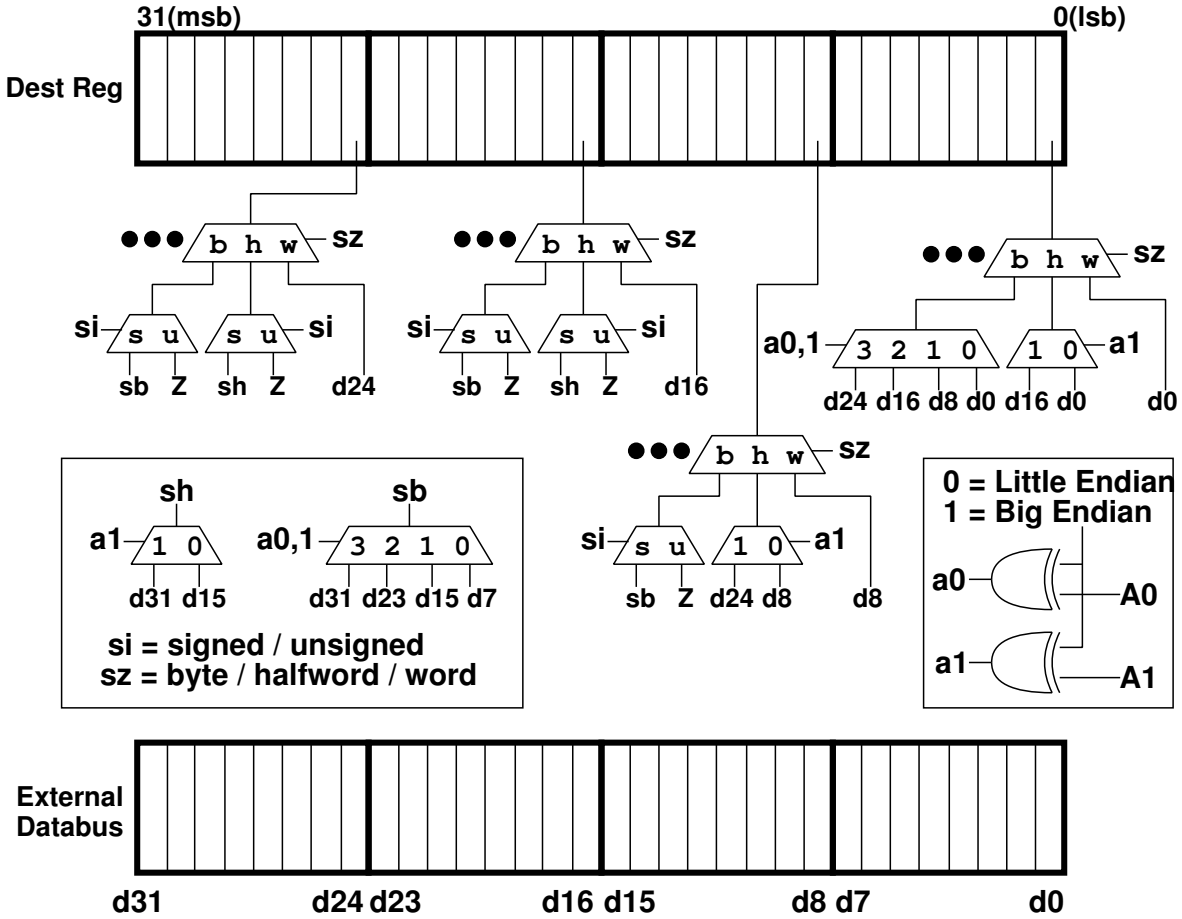
The rest of the system is normally swept up onto the same piece of silicon and this is denoted with the 'special function periperhal.' This would be the one part of the design that varies from product to product. The same core set of components would be used for all sorts of different products, from iPODs, digital cameras or ADSL modems.

Aligned Loads and Stores

- Address mod sizeof(type) = 0
- Most ISA support 8,16,32,(64) bit loads and stores in hardware
- Signed and unsigned stores same
- Sub-word loads can be Signed and Unsigned
  - CISC: loads merge into dest reg
  - RISC: loads extend into dest reg E.g:


increasing address
19
20
21
22
23
24
D
E
F
G
H
I
byte load from addr=21
32bit word
unsigned
result
31
0 (lsb)
0
0
0
F
signed
result
31
0 (lsb)
S
S
S
F

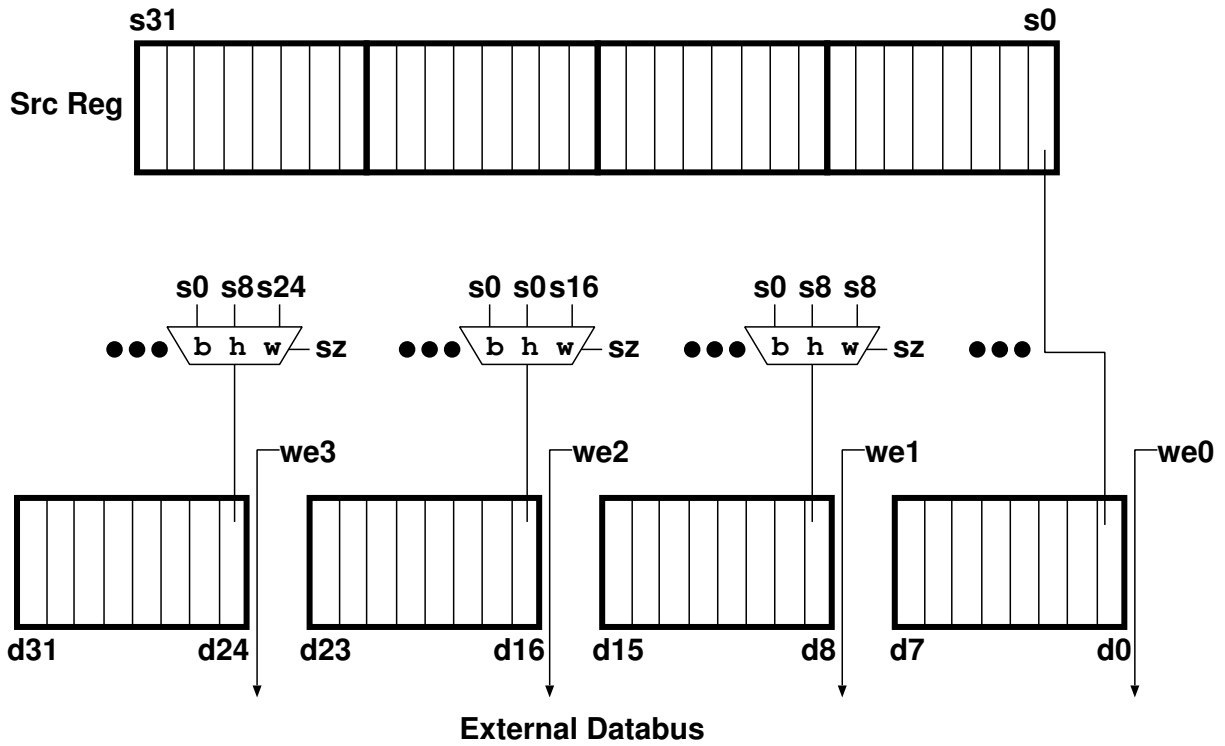
Aligned Sub-word Load Logic



- byte-lane steering
- sign/zero extension

- Big/Little endian modes

Aligned Sub-word Store Logic



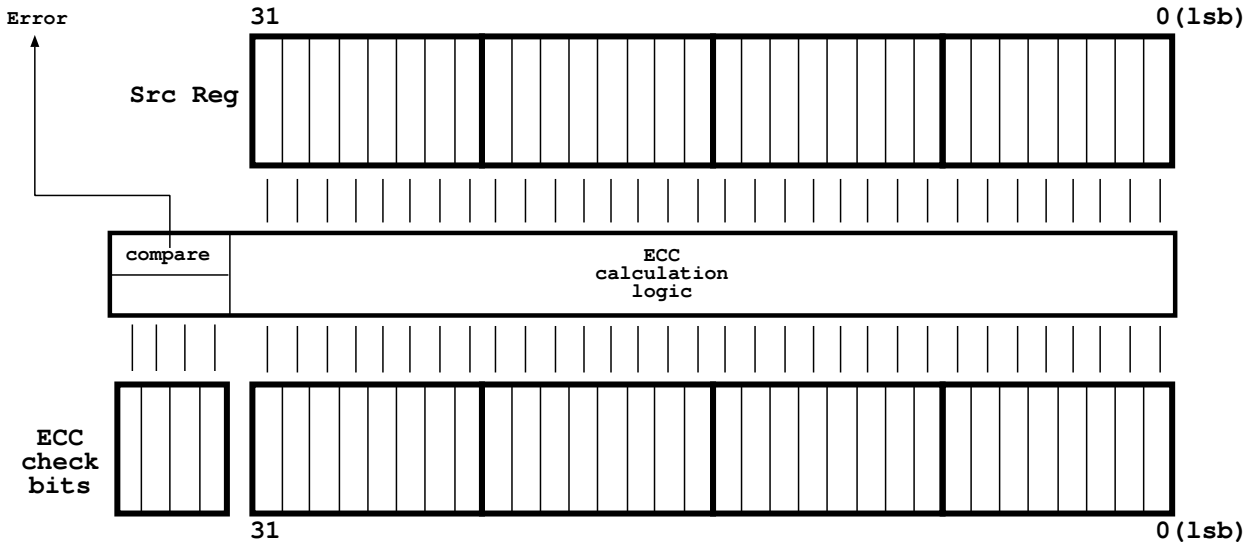
a1	a0	sz	we3	we2	we1	we0
0	0	w	1	1	1	1
0	0	h	0	0	1	1
1	0	h	1	1	0	0
0	0	b	0	0	0	1
0	1	b	0	0	1	0
1	0	b	0	1	0	0
1	1	b	1	0	0	0

- Replicate bytes/halfwords across bus

- Write enable lines tell memory system which byte lanes to latch

Sub-Word Load/Stores

- Word addressed machines
  - Addr bit A0 addresses words
- Alpha (v1):
  - Byte addressed, but 32/64 load/stores only
  - Often critical path
  - Sub-word stores hard with ECC memory
  - So, emulate in s/w using special instructions for efficiency


Error
31
0(lsb)
Src Reg
compare
ECC
calculation
logic
ECC
check
bits
31
0(lsb)

Emulating Byte Loads

1. Align pointer

2. Do word load

3. Shift into low byte

4. Mask

5. (sign extend)

- e.g. 32bit, Little Endian, unsigned

```
unsigned int temp;
temp = *(p&(~3));
```

```
temp = temp >> ((p&3) *8);
reg  = temp & 255;
```

- e.g. 32bit, Big Endian, unsigned

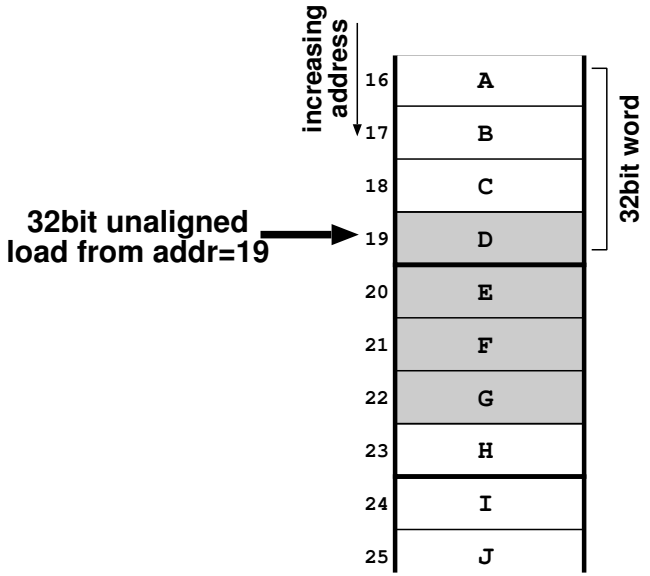
```
unsigned int temp;
temp = *(p&(~3));
temp = temp >> ( (3-(p&3)) * 8);
reg  = temp & 255;
```

- e.g. 64bit, Little Endian, signed

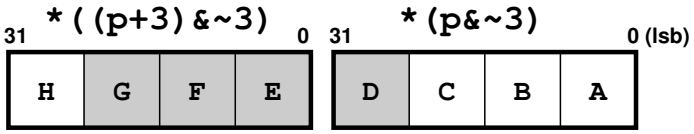
```
long temp;
temp = *(p&(~7));
temp = temp << ( (7-(p&7)) * 8);
reg  = temp >> 56;
```

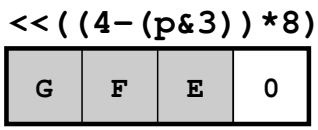
Unaligned Accesses

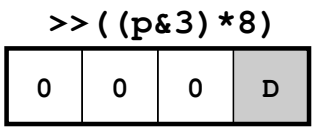
- Address mod sizeof(value) $\neq 0$
- E.g. :

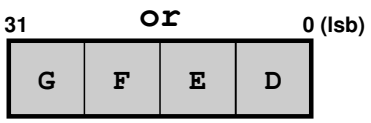


Little Endian



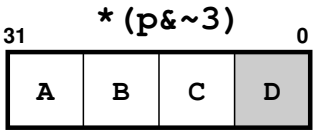


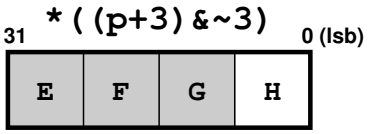


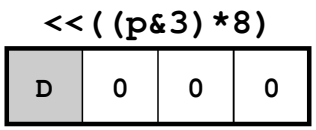


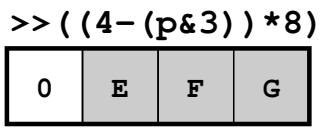
result

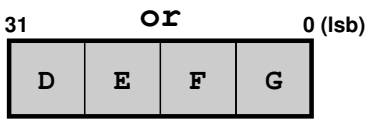
Big Endian











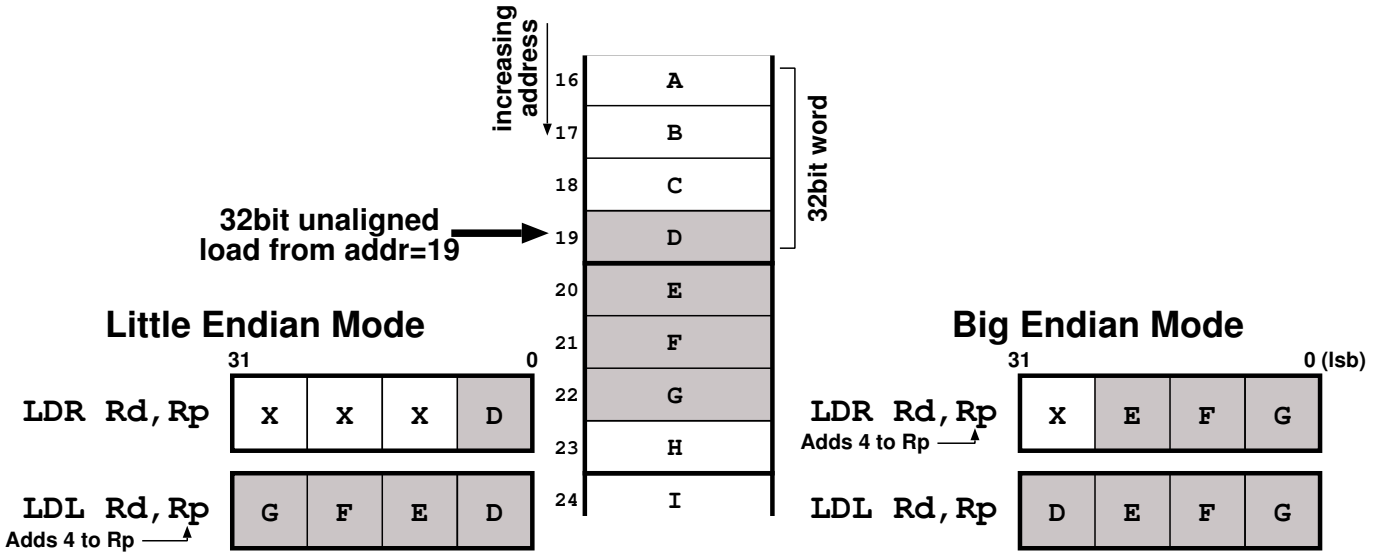
Unaligned Accesses

- CISC and Power PC support unaligned accesses in hardware
  - Two memory accesses
    - $*$ $\rightarrow$ Less efficient
  - May cross page boundaries
- Most RISCs synthesize in software
  - Provide special instructions
- Compilers try to keep data aligned
  - `struct` element padding
- Casting `char *` to `int *` dangerous

MIPS Unaligned Support

- `LWR` Load Word Right
- `LWL` Load Word Left
  - Only one memory access per instruction
  - Does shifting and merging as well as load

  → Unaligned load in 2 instrs



- `STR` Store Word Right
- `STL` Store Word Left
- Uses byte store hardware to merge into memory/cache

Alpha Unaligned Loads

- `LDQ` trap if not 8byte aligned
- `LDQ_U` ignore a0-a2
- `EXTQL Rd ← Rs, Rp`
  Shift Rs right by Rp&7 bytes and extracts quad word into Rd.
- `EXTQH Rd ← Rs, Rp`
  Shift Rs left by 8-Rp&7 bytes and extracts quad word into Rd.
- Alpha requires 5 instrs for arbitrary unaligned load

```
LDQ_U   Rd ← Rp
LDQ_U   Re ← Rp + #7
EXTQL   Rd ← Rd, Rp
EXTQH   Re ← Re, Rp
OR      Rd ← Rd, Re
```

- `EXTBL Rd ← Rs, Rp`
  Shift Rs right by Rp&7 bytes and extracts low byte into Rd.

- also `EXTLL, EXTLH, EXTWL, EXTWH`

- If alignment of pointer is known, may use optimized sequence
  E.g. load 4bytes from address 0x123
  `LDQ Rd ← -3(Rp)`
  `EXTLL Rd ← Rd, #3`

Alpha unaligned stores

- No byte hardware, so load quad words, merge, and store back

- `INSQL Rd ← Rs, Rp`
  Shift Rs left by Rp&7 bytes

- `INSQH Rd ← Rs, Rp`
  Shift Rs right by 8-Rp&7 bytes

- `MSKQL Rd ← Rs, Rp`
  Zero top 8-Rp&7 bytes

- `MSKQH Rd ← Rs, Rp`
  Zero bottom Rp&7 bytes

- E.g.: Store quad word Rv to unaligned address Rp

```
LDQ_U   R1 ← Rp        Load both quad words
LDQ_U   R2 ← Rp + #7
INSQH   R4 ← Rv, Rp    Slice & Dice Rv
INSQL   R3 ← Rv, Rp
MSKQH   R2 ← R2, Rp    Zero bytes to be replaced
MSKQL   R1 ← R1, Rp
OR      R2 ← R2, R4    Merge
OR      R1 ← R1, R3
STQ_U   R2 → Rp + #7   Store back
STQ_U   R1 → Rp        Order important:aligned case
```

Copying Memory

- Often important:
  - OS: user args, IPC, TCP/IP
  - user: realloc, pass-by-value
- `memmove`
  - Must deal correctly with overlapping areas
- `memcpy`
  - Undefined if areas overlap
  - Enables fixed direction
- `copy_aligned`

  - Source and Dest `long` aligned
  - Fastest

- Small copies (< 100 bytes)
  - Avoid large start-up costs

- Medium sized copies (100–100KB bytes)
  - Use highest throughput method

- Large copies
  - Probably memory b/w limited anyway...

copy_aligned

- E.g. for 32bit machine

```
void copy_aligned( int32 *d, const int32 *s, int n)
{
  sub n, n, #4
  blt n, return   ; if n<0 exit
loop:
  ldw tmp, (s)
  add d, d, #4
  sub n, n, #4    ; set branch value early
  add s, s, #4
  stw tmp, -4(d)  ; maximise load-to-use
  bgt n, loop     ; if n>0 branch (no delay slot)
}
```

- Use widest datapath
  - (64bit FP regs on PPro)
- Maximize cycles before tmp is used

- Update n well in advance of branch

- To further optimize:
  - Unroll loop to reduce loop overhead
  - Instruction scheduling of unrolled loop
  - (software pipelining)

copy_aligned (2)

```
void copy_8_aligned( int32 d[], const int32 s[], int n)
{
    int32 t0,t1,t2,t3,t4,t5,t6,t7;
top:
    t0 = s[0];   t1 = s[1];
    t2 = s[2];   t3 = s[3];
    t4 = s[4];   t5 = s[5];
    t6 = s[6];   t7 = s[7];
    n  = n - 32; s  = s + 32;
    d[0] = t0;   d[1] = t1;
    d[2] = t2;   d[3] = t3;
    d[4] = t4;   d[5] = t5;
    d[6] = t6;   d[7] = t7;
    d  = d + 32; if (n) goto top;
}
```

- Need to deal with boundary conditions
  - e.g. if n mod 32 != 0
- Get cache line fetch started early

- Issue a load for the next cache line
  - OK if non-blocking cache
  - beware exceptions (array bounds)

  $\Rightarrow$ prefetch or speculative load & check

  $\Rightarrow$ non-temporal cache hints

- IA-64: 'Rotating register files' to assist software pipelining without the need to unroll loops

Unaligned copy

- E.g. 32bit, Little Endian

```
void memcpy( char *d, const char *s, int n)
{
  uint32 l,h,k,*s1,*d1;

  /* Align dest to word boundary */
  while ( ((ulong)d&3) && n>0 ) {*d++ = *s++; n--;}

  /* Do main work copying to aligned dest */
  if( ((ulong)s & 3) == 0 ) {       /* src aligned ? */
     k = n & ~3;                    /* round n down  */
     copy_aligned(d, s, k);
     d+=k; s+=k; n&=3;              /* ready for end */
  }
  else
  {
    s1 = (uint32 *)((ulong)s & ~3); /* round s down  */
    d1 = (uint32 *) d;              /* d is aligned  */
    h  = *s1++;                     /* init h        */
    k  = (ulong)s &3;               /* src alignment */
    for(; n>=4; n-=4) {             /* stop if n<4   */
      l = *s1++;
      *d1++ = ( h >> (k*8)     ) |
              ( l << ((4-k)*8) ) ;
      h = l;
    }
    d = (char *) d1;                /* ready for end */
```

```
    s = ((char *)s1) - 4 + k;
  }

  /* Finish off if last 0-3 bytes if necessary */
  for( ; n>0; n-- ) *d++ = *s++;
}
```

Memory Translation and Protection

- Protection essential, even for embedded systems
  - isolation, debugging
- Translation very useful
  - demand paging, CoW, avoids relocation
- Segmentation *vs.* Paging
  - x86 still provides segmentation support
  - descriptor tables: membase, limit
  - segment selectors : cs, ds, ss, fs, gs
- Page protection preferred in contemporary OSes

- Translation Lookaside Buffer (TLB)
  - translate Virtual Frame Number to PFN
  - check user/supervisor access
  - check page present (valid)
  - check page writeable (DTLB)
- Separate D-TLB and I-TLB
  - often a fully associative CAM
  - separate I-TLB and D-TLB
  - typically 32-128 entries
  - sometimes an L2 Joint-TLB e.g. 512 entry
- Hardware managed *vs.* software managed TLB

Hardware page table walking

- Hierarchical lookup table

- E.g. x86/x86_64 4KB pages evolved over time:
  - 2-level : 4GB virt, 4GB phys (4B PTEs)
  - 3-level : [512GB] virt, 64GB phys (8B PTEs)
  - 4-level : 256TB virt, 1TB phys (8B PTEs)
    (48 bit VAs are sign extended to 64bit)

- 'set PT base' instruction
  - implicit TLB flush (on x86)

- Flush virtual address

- Global pages not flushed
  - special bit in PTE
  - should be same in every page table!
  - typically used for kernel's address space
  - special TLB flush all
- Superpages are PTE 'leaves' placed in higher levels of the page table structure
  - e.g. 4MB pages on x86 2-level

Software managed TLB

- OS can use whatever page table format it likes
  - e.g. multilevel, hashed, guarded, etc.
  - (generally more compact than hierarchical)
  - use privileged 'untranslated' addressing mode
- Install TLB Entry instruction
  - specify tag and PTE
  - replacement policy usually determined by h/w
    - e.g. not most recently used
- (may allow TLB contents to be read out for performance profiling)

- Flush all, flush ASN, flush specified VA
- Flexible superpage mappings often allowed of e.g. 8, 64, 512 pages.
- Notion of current Address Space Number (ASN)
- TLB entries tagged with ASN
- Try to assign each process a different ASN
  - no need to flush TLB on process switch
  - (only need to flush when recycling ASNs)
- IA-64 : s/w TLB with hardware PT walking assist
- PPC: h/w fill from larger s/w managed hash table

ISA Summary

- RISC
  - Product of quantitative analysis
  - Amdahl's Law
  - Load-Store GPRs
  - ALU operates on words
  - Relatively simple instructions
  - Simple addressing modes
  - Limited unaligned access support
  - (s/w managed TLB)

- Architecture extensions
  - Backwards compatibility
- Copying memory efficiently

Does Architecture matter?