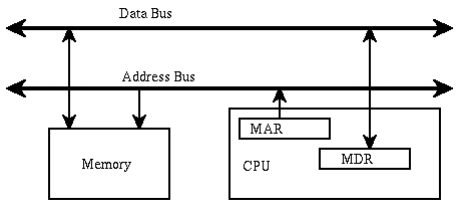
CPU Design

- Review: from 31V4
  - CPU/Memory Interconnection

We now propose to study the CPU itself: how it executes instructions and how this is controlled, and then how to make the CPU run faster.



We will start with a simple CPU, then complicate matters.

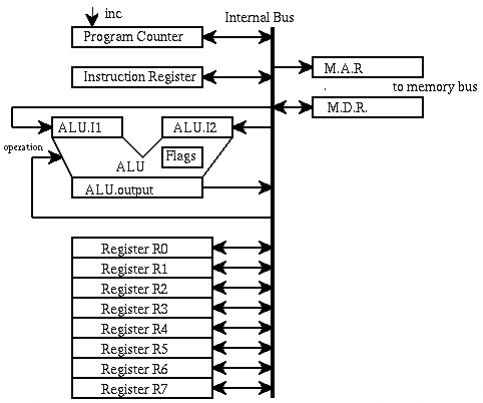
Copyright 1998, 2004 © Leslie S. Smith

A (very) Simple CPU

- Simple single-bus CPU.

Registers:
PC: Program Counter
IR: Instruction Register
ALU: Arithmetic and Logic Unit
MDR: memory data register
MAR: memory address register

- Single internal bus
  - provides communication between all the internal registers



Copyright 1998, 2004 © Leslie S. Smith

How does instruction execution work?

- Instruction at address in PC is fetched to IR
  - Register transfers…
- IR is decoded:
  - The bit pattern inside the IR is used
    - To control the transfer of data from CPU register to CPU register
    - To set up the operation inside the ALU
    - To determine the operation of reading or writing to or from memory
- Actual operation is a sequence of transfers or ALU operations
  - The end result is the execution of the instruction
- (Example)

Copyright 1998, 2004 © Leslie S. Smith

Arithmetic and Logic Unit

- All data alteration takes place here
- Operation:
  - Data placed on input registers (ALU.I1, ALU.I2)
  - Operation code placed in ALU

  ] Generally Simultaneously
  - After a brief delay, result appears on output register, ALU.output
- ALUs are complex
  - Contain all logic necessary to perform
    - arithmetic operations

    increment, decrement, add, add with carry, subtract, subtract with borrow, multiply and divide (signed and unsigned)
    - logical operations

    negate, and, or, eor
    - shift operations

    shift left (logical and circular), shift right (logical and circular)
  - Higher-performance CPUs have an additional floating-point ALU
    - performs all floating-point operations

Copyright 1998, 2004 © Leslie S. Smith

Register Transfer Language

- RTL is used to describe transfers between registers (etc) inside the CPU

e.g.

R1 <- R2

means that R1 receives the value in R2. (R2 is unaltered)

- More than one transfer can take place at a time:

e.g.

R1<-R2 R3 <- R4

means that R1 receives the value in R2 at the same time as R3 receives the value in R4

- Sometimes, not all the bits are transferred

R1 <- R2#0x0000FFFF

  - only the least significant 16 bits are transferred (R2 is masked (AND'd) with hex0000FFFF)

Copyright 1998, 2004 © Leslie S. Smith

In the simple CPU

- all transfers use the bus, so
- only one transfer can take place at a time
  - strictly speaking, one value could be read into several registers concurrently, but this is virtually never required

Copyright 1998, 2004 © Leslie S. Smith

Implementing Instructions

- Consider the instruction, which is located at address 0x3FFE0124

  LD.W R2, 0x134C ; Load Register 2 with the Word at location 0x134C
- We assume
  - the PC already points at 0x3FFE0124 (byte address)
  - the operand (0x134C) is the last 16 bits of the instruction
- First we fetch the instruction

  MAR <- PC++

  MDR <- Memory[MAR]

  IR <- MDR
  - Note that we assume that we can increment the PC without transferring it to the ALU and performing an increment operation on it. Generally the processor will have special hardware to increment the PC.
  - The earliest processors did not!

Copyright 1998, 2004 © Leslie S. Smith

Executing the instruction

- Now we can decode the instruction (more about this later).
- This results in the following operations

  MAR <- IR#0x0000FFFF // lowest 16 bits

  MDR <- Memory[MAR]

  R2 <- MDR
  - We assume the whole instruction was in one 32 bit word, so we load the part of the instruction that contains the address into the MDR, then run another memory access cycle to retrieve the data at that address.
  - Then we transfer the data to R2
- That's all: we go on to the next instruction (at 0x3FFE0128)

Copyright 1998, 2004 © Leslie S. Smith

A More Complex Instruction

- Let's say that at location 0x3FFE0128 we have the instruction
  ADD.W R1, R2 ; ADD R2 to R1, leaving result in R1
- Start (again) by fetching the instruction
  MAR <- PC++
  MDR <- Memory[MAR]
  IR <- MDR
- Then decoding the instruction results in putting the operands into the ALU registers:
  ALU.I1 <- R1
  ALU.I2 <- R2
- placing the operation into the ALU operation register
  ALU.operation <- IR#BITS
  - The ALU operation code will be inside the instruction: masking will be required
- and storing the result (after an appropriate delay)
  R1 <- ALU.output

Copyright 1998, 2004 © Leslie S. Smith

A Complex Instruction

- CISC instructions are often more complex than the above. Consider
  ADD.W R1, 123(R2) ; add the contents of the word 123 offset from where R2 points to
  ; R1, leaving the result in R1
- Fetch…
  MAR <- PC++
  MDR <- Memory[MAR]
  IR <- MDR
- then decoding the instruction results in computing the effective address (EA) or operand address
  ALU.I1 <- R2
  ALU.I2 <- IR#00000FFF // let's say 12 Least Significant 12 bits are used for offsets
  (ALU add cycle)
- The EA is now in ALU.output. So we can fetch the operand
  MAR <- ALU.output
  MDR <- Memory[MAR]

Copyright 1998, 2004 © Leslie S. Smith

A Complex Instruction (continued)

- We can now execute the operation required

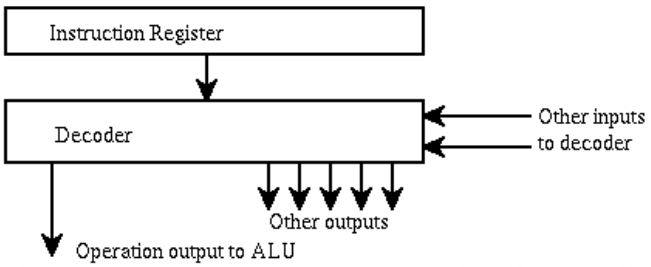
  ALU.I1 <- MDR
  ALU.I2 <- R1
  ALU.operation <- IR#BITS (ALU add cycle)
  R1 <- ALU.output

- Such instructions
  - take a long time
  - need careful sequencing
    - this means that the decoding circuitry needs to be complex

Copyright 1998, 2004 © Leslie S. Smith

Making Instructions Faster

- One simple improvement is not to use one bus for everything
- e.g. pass the ALU operation code to the ALU using a dedicated path
- This simple operation can save a small number of cycles.
  - Because one operand and the operation can be set up simultaneously



- But real impr rent approach (e.g. RISC instructions)
- We will look at both options.

Copyright 1998, 2004 © Leslie S. Smith

RISC Instructions

- On most RISC machines, the primary instructions are
  - Load and Store instructions
  - Data altering instructions (using the ALU)
- Loads and Stores use 1 register and 1 simple effective address
  - LD R1, R2 % load register 1 from register 2
  - LD R1, #0xF5667F3 % load register with value (immediate)
  - LD R1, 30(R4) % register indirect with offset
- Data altering instructions use ONLY registers
  - SUB R1, R2, R3 % R1 := R2 – R3
  - AND R3, R4, R5 % R3 := R4 AND R5
  - (most RISC architectures use 3 operand instructions for these)
- Branch and Jump and Jump to subroutine instructions use only simple addressing modes: PC relative (PC indirect with offset), direct

Copyright 1998, 2004 © Leslie S. Smith