Prequel : Designing a digital CPU-based Computer

Things to consider first:

1 — what is your ALU operands' bit width?

2 — what is the number of ALU operations you'd like to support?

3 — then, what is the bit width of your operational codes, "aka opcodes", which will determine the actions your CPU will perform, e.g., add, subtract, multiply, divide, load from memory into registers, save from registers to memory, branching, etc?

4 — what is the number of registers you will provide the software developer to program your computer?

5 — do you want a dedicated register to receive the results of the ALU? This is called an "accumulator."

6 — what are the supported instruction formats? memory-reference (immediate-type or "i-type"), arithmetic-logic (register-type or "r-type"), branch (jump-type or "j-type"). How many registers do you want to provide in the r-type? Two (accumulator and a target register)? Three (target, source-1, source-2)?

7 — based upon the bit widths of the ALU operands, opcode, instruction types, number of registers, then what will you define as the instruction width? Will that width be fixed or dynamic? Remember of design principles! RISC or CISC or combination of both, or not? :)

8 — how will you design your datapath, control unit in order to implement your instruction set architecture (ISA)?

9 — will the datapath & control unit be implemented using single-cycle instruction execution? How will you implement? MINIMUM REQUIREMENT FOR YOUR FINAL PROJECT TO PASS. Do the best you can. Ask for assistance.

10 — will the single-cycle implementation be converted to a pipelined design? How many stages? How will you implement it? (OPTIONAL BUT WILL GAIN MUCH EXTRA CREDIT if working.)

11 — What digital building blocks will you need to implement your processor design in Verilog? How will you integrate these blocks together? Will they only be behaviorial models? (that's ok! :)) See a growing list here

https://github.com/robmarano/ece251_at_cooper/tree/ottobit_cpu

Designing a CPU-based Computer

I - Recall the underlying principles of computer HW design:

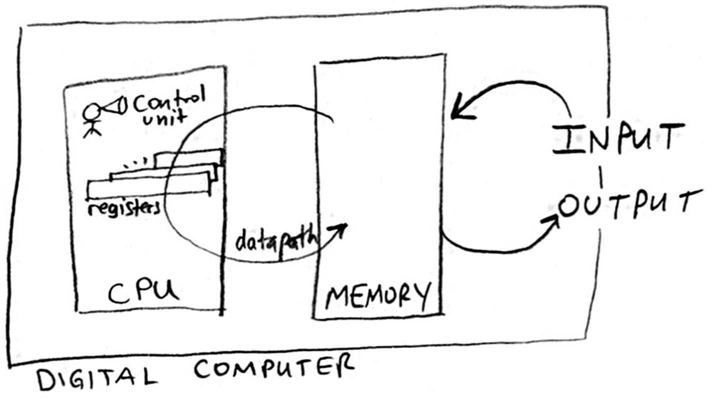
D.P.1 - Simplicity favors regularity ⇝ (ex) instructions w/ consistent # of operands → easier in HW // i-bit width instructions

D.P.2 - Smaller is faster ⇝ (ex) fewer regs, faster to access & retrieve data

D.P.3 - Make the common case fast ⇝ (ex) simple, common instructions (small #) → faster in HW

D.P.4 - Good design demands good compromise ⇝ (ex) compromise on #1 to introduce small # of supported instruction formats.

II Big picture of a digital computer; von Neumann (Princeton) architecture using stored-program concept



III Byte-Order

- big endian ≡ use addr of left-most byte (of data) as the mem addr
- little endian ≡ use addr of right-most byte (of data) as the mem addr

(ex)

Ø	1	2	3

addr (at 3)

3	2	1	Ø

addr (at Ø)

IV n-bit CPU ⇒ n ≡ bit width of ALU operands

(ex) 4-bit CPU ⇒ 4bit operands for $\underbrace{\text{ADD, SUB, MUL, DIV}}_{\text{arithmetic}}$,..., $\underbrace{\text{NOR, AND, OR, Shifts (logical \& arith)}}_{\text{logical}}$

V MEMORY amount & width; addressable

m-bit wide memory "addressable" ⇒ if m = n, "good chance" to move into/out of mem in the least amount of clock cycles

if m > n, "most probably" need many additional cycles to move data into/out of mem

if n = 4 bits (ALU operands sized @ 4 bits)
m = 12 bits (mem addr sized @ 12 bits)
} 2^{12} addressable mem locations = 4Ø96 & each location has 4 bits of data.

(modeled against a true RISC CPU "MIPS")

VI instruction width & formats

recall: n-bit sized operands
m-bit sized mem locations } i-bit sized instructions will need to support functionalities that can:

(a) load & store data from/to mem;
(b) perform arithmetic-logical calculations;
(c) jump to other code blocks & return

by D.P.1 ⇒ set instruction width to "i"
but what value is i?

* variable width instructions add too much HW complexity.

instruction formats

* simplicity would encourage a single instruction format BUT too restrictive ∴ use D.P.4.

one compromise: set # of instruction formats to a low & manageable #, i.e., as presented above:

aka

mem-reference → (a) I-type "immediate" → operate on 2 registers and an immediate value

arithmetic-logical → (b) R-type "register" → operate on 3 registers (or perhaps 2 regs and an ACC)

branching → (c) J-type "jump" or branching → operate on one immediate value

R-type instruction format "register type"

- in modern, general-purpose CPUs → use 3 registers as operands : two sources; one destination
- the R-type instruction may have 6 fields:

fields 0 & 5 { what operation the instruction performs { op ≡ operation code (opcode)
funct ≡ function

all R-type instrs have opcode ≡ ∅
the specific R-type operation ≡ funct

fields 1,2,3 { operands encoded in three fields;
rs ≡ source register
rt ≡ another source register
rd ≡ destination register

field 4 { the 5th field ≡ "shamt" for shift operations; otherwise shamt = ∅
shamt ≡ shift amount

---

instruction alignment to memory could be:
- by byte
- by word
⋮

I-type instruction format

- in modern, general-purpose CPUs $\rightarrow$ use 2 register operands & $\underbrace{\text{1 immediate value}}_{\text{bit size determined based upon available bits in instr.}}$
- the I-type instr may have 4 fields:

  op ≡ opcode $\rightarrow$ determines functionality/operation

  rs ≡ source reg

  rt ≡ destination reg for some opcodes & source for others; ex: addi, lw vs. sw

  imm ≡ source as immediate value) bit width TBD
  $\hookrightarrow$ in 2's complement format)
  $\hookrightarrow$ this value can be sign-extended for arithmetic ops & zero-sign extend for logical ops

(ex) $5_{10} = Ø1Ø1_2$

↓ sign extend to 8 bits

ØØØØ Ø1Ø1

$-5 = -(Ø1Ø1) \rightarrow$ 1Ø1Ø + 1 = 1Ø11

= 1Ø11

↓ sign extend to 8 bits

1111 1Ø11

↓ convert to #

$-(1111\ 1Ø11) \rightarrow$ ØØØØ Ø1ØØ + 1 = ØØØØ Ø1Ø1

J-Type instruction format

- in modern, general-purpose CPUs, $\rightarrow$ to support decisions in code, able to jump to other code blocks based on conditional tests
- in non-branching code $\rightarrow$ the Program Counter (PC) advances to next instruction
  $\rightarrow$ the distance (bit width) to the next instruction is based on the length of an instruction (which has constant width)
- branching is either a
  - conditional branch $\rightarrow$ to the expected instr @ a mem addr
  - unconditional branch (aka jumps) $\rightarrow$ has several versions
    - jump directly to instr at label (mem addr)
    - jump and link : similar to jump and used by functions to save a return addr
    - jump register : <R-type instr> jumps to an addr stored in a register. This addr uses the entire bit width of a register INSTEAD of partial as used in jump and link which uses an offset from the PC

Addressing Modes

- modern, general-purpose CPUs may have many addressing modes, including but not limited to:

(a) register-only addressing :=
- uses regs for all source & dest operands
- all R-type instrs use reg-only addressing

(b) immediate addressing :=
- uses "x-bit" immediate values along w/ regs as operands.
- some I-type instrs use this addressing; (ex) ADDI, LUI in MIPS

(c) base addressing :
- mem access instrs use this; (ex) LOAD, STORE.
- effective addr of mem operand = base addr in rs to sign-extended offset in imm.

(d) PC-relative addressing :
- Conditional branch instructions use this to specify the new value of the PC if branch taken (branch target addr) BTA
- the signed offset in the immediate field is added to PC to obtain new PC
- hence, the branch destination addr is relative to the current PC.

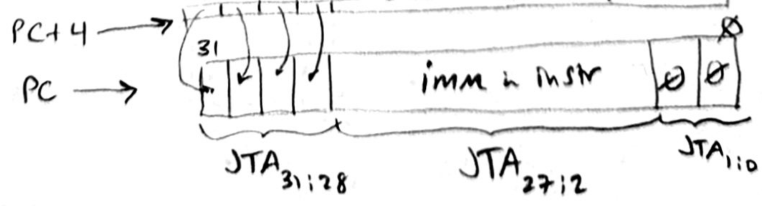
(e) pseudo-direct addressing :
- in direct addressing ⇒ addr is specified in the instr.
- jump and jump-and-link instructions would use direct addressing to specify a m-bit jump target addr (JTA) to indicate the instr addr to execute next.
- BUT J-Type instrs' encoding does not have enough bits to specify a full m-bit JTA.
  - 6 bits used for opcode
  - so m-6 bits left to encode JTA
  - because instructions are m-bit aligned, there is a chance some of the least significant bit(s) would be ∅.

from MIPS (ex) if instruction is 32-bits wide ⇒ word aligned ∴ $JTA_{1:0}$ 2 least significant bits = ∅

6 bits = opcode
26 bits = for JTA

hence $JTA_{27:2}$ ← taken from the immediate value in the instr.

$JTA_{31:28}$ ← taken from the four most-significant bits of PC + 4 → b/c word aligned → 32 bits = 4 bytes



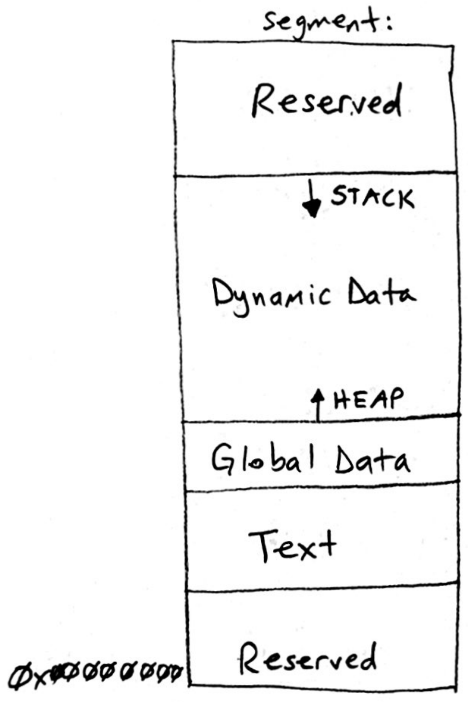
VII Memory map

- with m-bit addresses $\longrightarrow$ address space spans 2^m bytes

  if (ex) $m = 12$ bits

  $\therefore$ address space spans 2^{12} bytes $= 4096$ bytes

- Mem map diagram



segment:
Reserved
STACK
Dynamic Data
HEAP
Global Data
Text
Reserved
0x00000000