Lecture 6

Decomposing Y86-84 Instructions

In this lecture we will begin to develop a sequential Y86 processor, examining how the various instructions can be represented in a consistent form which will ultimately be implemented by logic circuits.

Decomposing Instructions into Stages

The first step to implementing the Y86 instruction set architecture is to decompose instructions into a standard set of stages which are consistent across all instructions in the set.

Fetch: Read the instruction bytes from the instruction memory, using the program counter (PC) as the address. It extracts two 4-bit components from the instruction byte: the instruction code, `icode`, and the instruction function, `ifun`. If appropriate, it also fetches the register specifier byte, to determine `rA` and `rB`, and also possibly the 8-byte constant, `valC`. Finally, it calculates the address of the next instruction in `valP`.

Decode: Read up to two operands from the register file, giving `valA`, and/or `valB`.

Execute: Use the arithmetic logic unit (ALU) to perform the operations indicated by the value of `ifun`, calculate an effective address, or update the stack pointer. The result of these operations is `valE`. The condition codes are updated to determine if a conditional move or a conditional jump should be executed.

Memory: Write data to memory, or read data from memory into `valM`.

Write Back: Write up to two results back to the register file.

PC Update: Update PC to the address of the next instruction.

Smaller and more consistent the stages will lead to a simpler hardware design. Note also that the ALU is used for three different purposes: performing arithmetic and logical operations on registers, calculating effective address and updating the stack pointer. Sharing one piece of hardware across multiple purposes minimises the total size of the hardware design.

Y86-64 Instructions

The first set of instructions all result in a new value being written to register `rB`. The source of this value is either an operation applied to the `rA` and `rB` registers, the content of the memory addressed by `rA`, or the immediate value encoded in the instruction.

Stage	`OPq rA,rB`	`rrmovq rA,rB`	`irmovq V,rB`
Fetch	`icode:ifun` $\leftarrow M_1$[`PC`] `rA:rB` $\leftarrow M_1$[`PC`+1] `valP` $\leftarrow$ `PC`+2	`icode:ifun` $\leftarrow M_1$[`PC`] `rA:rB` $\leftarrow M_1$[`PC`+1] `valP` $\leftarrow$ `PC`+2	`icode:ifun` $\leftarrow M_1$[`PC`] `rA:rB` $\leftarrow M_1$[`PC`+1] `valC` $\leftarrow M_8$[`PC`+2] `valP` $\leftarrow$ `PC`+10
Decode	`valA` $\leftarrow R$[`rA`] `valB` $\leftarrow R$[`rB`]	`valA` $\leftarrow R$[`rA`]	—
Execute	`valE` $\leftarrow$ `valB OP valA` Set `CC`	`valE` $\leftarrow 0 +$ `valA`	`valE` $\leftarrow 0 +$ `valC`
Memory			
Write Back	R[`rB`] $\leftarrow$ `valE`	R[`rB`] $\leftarrow$ `valE`	R[`rB`] $\leftarrow$ `valE`
PC Update	`PC` $\leftarrow$ `valP`	`PC` $\leftarrow$ `valP`	`PC` $\leftarrow$ `valP`

The second set of instructions also involve a memory stage. They both read a base address from the instruction, and then use the ALU to calculate an effective address, using register `rB` as an offset. Then, either the contents of register `rA` are written to this address, or the contents of this address are written to register `rA`.

Stage	`rmmovq rA,D(rB)`	`mrmovq D(rB),rA`
Fetch	`icode:ifun` $\leftarrow M_1$[`PC`] `rA:rB` $\leftarrow M_1$[`PC`+1] `valC` $\leftarrow M_8$[`PC`+2] `valP` $\leftarrow$ `PC`+10	`icode:ifun` $\leftarrow M_1$[`PC`] `rA:rB` $\leftarrow M_1$[`PC`+1] `valC` $\leftarrow M_8$[`PC`+2] `valP` $\leftarrow$ `PC`+10
Decode	`valA` $\leftarrow R$[`rA`] `valB` $\leftarrow R$[`rB`]	`valB` $\leftarrow R$[`rB`]
Execute	`valE` $\leftarrow$ `valB` $+$ `valC`	`valE` $\leftarrow$ `valB` $+$ `valC`
Memory	M_8[`valE`] $\leftarrow$ `valA`	`valM` $\leftarrow M_8$[`valE`]
Write Back		R[`rA`] $\leftarrow$ `valM`
PC Update	`PC` $\leftarrow$ `valP`	`PC` $\leftarrow$ `valP`

The next two instructions use the ALU to update the stack pointer, subtracting or adding 8 bytes as appropriate. Note that `popq rA` writes back two values to the register file.

The next set of instructions all make more complex updates to the program counter, `PC`, than

Stage	`pushq rA`	`popq rA`
Fetch	`icode:ifun` $\leftarrow M_1$[`PC`] `rA:rB` $\leftarrow M_1$[`PC`+1] `valP` $\leftarrow$ `PC`+2	`icode:ifun` $\leftarrow M_1$[`PC`] `rA:rB` $\leftarrow M_1$[`PC`+1] `valP` $\leftarrow$ `PC`+2
Decode	`valA` $\leftarrow R$[`rA`] `valB` $\leftarrow R$[`%rsp`]	`valA` $\leftarrow R$[`%rsp`] `valB` $\leftarrow R$[`%rsp`]
Execute	`valE` $\leftarrow$ `valB` + (-8)	`valE` $\leftarrow$ `valB` + 8
Memory	M_8[`valE`] $\leftarrow$ `valA`	`valM` $\leftarrow M_8$[`valA`]
Write Back	R[`%rsp`] $\leftarrow$ `valE`	R[`%rsp`] $\leftarrow$ `valE` R[`rA`] $\leftarrow$ `valM`
PC Update	`PC` $\leftarrow$ `valP`	`PC` $\leftarrow$ `valP`

those above. Note that `jXX` makes use of the condition codes, `CC`, and `ifun` to determine whether the conditional jump should be made, and updates `PC` appropriately. `call` updates `PC` to the address encoded in the instruction, but must also decrement the stack pointer, and write the return address (the address of the next instruction) into the memory address indicated by the newly updated stack pointer. Conversely, `ret` must retrieve the return address from the memory address indicated by the stack pointer, and must increment the stack pointer. To do so, it must use two copies of the stack pointer: one in `rA` and one in `rB`.

Stage	`jXX Dest`	`call Dest`	`ret`
Fetch	`icode:ifun` $\leftarrow M_1$[`PC`] `valC` $\leftarrow M_8$[`PC`+1] `valP` $\leftarrow$ `PC`+9	`icode:ifun` $\leftarrow M_1$[`PC`] `valC` $\leftarrow M_8$[`PC`+1] `valP` $\leftarrow$ `PC`+9	`icode:ifun` $\leftarrow M_1$[`PC`] `valP` $\leftarrow$ `PC`+1
Decode		`valB` $\leftarrow R$[`%rsp`]	`valA` $\leftarrow R$[`%rsp`] `valB` $\leftarrow R$[`%rsp`]
Execute	`Cnd` $\leftarrow Cond$(`CC`,`ifun`)	`valE` $\leftarrow$ `valB` + (-8)	`valE` $\leftarrow$ `valB` + 8
Memory		M_8[`valE`] $\leftarrow$ `valP`	`valM` $\leftarrow M_8$[`valA`]
Write Back		R[`%rsp`] $\leftarrow$ `valE`	R[`%rsp`] $\leftarrow$ `valE`
PC Update	`PC` $\leftarrow$ `Cnd` ? `valC` : `valP`	`PC` $\leftarrow$ `valC`	`PC` $\leftarrow$ `valM`

Finally, the `cmovXX` instruction generalises the `rrmovq` instruction, using `CC` and `ifun` to determine whether the write back stage should be executed. See the problem sheet for its decomposition.

Note that all these decompositions follow the same principle that the processor does not need to read back the state updated by the instruction in order to complete the processing of the instruction. Rather than doing so the decomposition will typically generate the necessary signal and then use the signal multiple times. For example, the `call` instruction generates the decremented stack pointer in `valE`, and uses this to both update the stack pointer, and to write the return address to the stack, on the completion of the instruction. Furthermore, although we've described the sequence of calculations as sequential, note that they actually all act simultaneously, starting on the rise of each clock signal, reaching an equilibrium state sometime before the next clock signal.

Further Reading

The decompositions described here also appear in Chapter 4 of the book below:

> Computer Systems: A Programmer's Perspective
> Randal E. Bryant and David R. O'Hallaron, Pearson, 2010.

A similar decomposition (in less detail) appears in Chapter 4 of the textbook below for a MIPS processor.

> Computer Organization and Design: The hardware/software interface
> D A Patterson & J L Hennessy, Morgan-Kaufmann, 2013.