03 – The Junior Processor

Oscar Gustafsson

Designing a minimal instruction set

- What is the smallest instruction set you can get away with while retaining the capability to execute all possible programs you can encounter?

Designing a minimal instruction set

- What is the smallest instruction set you can get away with while retaining the capability to execute all possible programs you can encounter?
- Answer: One instruction is sufficient:
- `SBNZ a,b,c,d`
  - `PM[c] = PM[a]-PM[b]`
  - Branch to address `d` if result is not zero
  - Otherwise, continue to next instruction
- (There are other possible instructions as well.)

Designing a minimal instruction set

- This processor is obviously going to be slow
- Only of theoretical interest (c.f. Turing Machine)

Discussion break

- What is a small instruction set that you can live with?
- **Hint:** C programs should run reasonably efficiently on this machine

Result from discussion

- Instructions:
  - add, sub, mult, (divide, modulo)
  - and, or, xor, left/right shift (arithmetic/logic)
  - load, store, move
  - conditional/unconditional branches
  - call subroutine/return from subroutine, (return from interrupt)
- Addressing modes:
  - Direct
  - Immediate
  - Indexed
  - (Post/pre increment/decrement)

The Junior processor

- First try: Start with the operations available in C
- Goal: ASIP DSP for real-time iterative computing
- Assumption: RISC-like instructions
  - move-load-store, ALU/MAC, and program flow control (very similar to the list you produced last lecture)

Instruction classification

Instruction group/type	Operands	Operations	Mathematical description	Flags	Clock cycles
Load, store, and move	Register name	Data transfer	`DST(ADR)<=SRC(ADR)`	No flag	1
ALU instructions	Register names, or immediate data	Arithmetic and	`OpW <= OpA op OpB`	ALU Flags	1
Flow control	Way to get the target address	Jump taken decision	`if(condition)` `PC <= target`	No flags	1 or 3

Move-load-store instructions

- RISC processor: Simple architecture
- Data and parameters of a subroutine are loaded to the register file first
- Operands come either from register file or from immediate data (carried by an instruction)
- Results in the register file need to be written back to the data memory

Move-load-store instructions

Mnemonic	Operands	Description	Operation	Cycles
load	OpW, DA	Load data from mem 0/1	`OpW<= DM(DA)`	1
store	DA,OpA	Store data to mem 0/1	`DM(DA) <= OpA`	1
move	OpW,OpA	Copy between two registers	`OpW <= OpA`	1
move	OpW,imm	Copy immediate to registers	`OpW <= imm`	1

Addressing modes

Name	DA	Algorithm
Direct	D	16-bit constant as the direct memory address
Register indirect	R	A register containing the memory address
Post increment	$R++$	R gives the address, $R = R + 1$ after addressing
Pre decrement	$--R$	$R = R - 1$ before addressing, R gives address
(Immediate)	I	16-bit constant as the value, used in move insn.

Arithmetic instructions

- Basic arithmetic operations in C: add, subtract, multiply, division, and modulo
  - Division operation can usually be avoided by for example multiplying with the reciprocal. Otherwise it is fairly rare, implement it using a subroutine instead.
  - Modulo operation is used even more rarely for DSP arithmetic computing, we should implement it using a subroutine

Arithmetic instructions

Mnemonic	Operands	Operation	Flags
ADD	OpA, OpB	`OpW <= OpA + OpB`	C,Z,N,V
SUB	OpA, OpB	`OpW <= OpA - OpB`	C,Z,N,V
ABS	OpA, OpB	`OpW <= ABS(OpA)`	Z,N,V
INC	OpA, OpB	`OpW <= OpA + 1`	C,Z,N,V
DEC	OpA, OpB	`OpW <= OpA - 1`	C,Z,N,V
MPL	OpA, OpB	`A <= OpA * OpB`	Z,N,V
MAC	A,OpA, OpB	`A <= A + OpA * OpB`	Z,N,V
RND	A	`OpW <= SAT(ROUND(A))`	Z,N,V
CAC	A	`A <= 0`	Z,N,V

Note: MAC, RND, and CAC areb operating on the wide accumulator register. (Not a typical RISC instruction)

Logic and shift operations

Mnemonic	Operands	Operation	Flags
AND	OpA, OpB	`OpW <= OpA & OpB`	N,V,Z
OR	OpA, OpB	`OpW <= OpA | OpB`	N,V,Z
NOT	OpA	`OpW <= ~OpA`	N,V,Z
XOR	OpA, OpB	`OpW <= OpA ^ OpB`	N,V,Z
LS	OpA, OpB	`OpW <= OpA << OpB[3:0]`	N,V,Z
RS	OpA, OpB	`OpW <= OpA >> OpB[3:0]`	N,V,Z

Note: The C standard allows shifts to be implemented like this. That is, the following program has undefined behavior:

```
uint16_t a,b;
a = 12345;
b = 19;
a = a >> b; // Undefined, shifting more than 16 bits
            // on a 16-bit datatype
```

Logic operators in C

<	Less than	!=	Not equal to
<=	Less than or equal to	&&	Boolean AND
==	Equal to	||	Boolean OR
>=	Greater than or equal to	!	Boolean NOT
>	Greater than		

- Do we need special ALU instructions for these?
  - Probably not, we can handle these by conditional branch instruction
  - (Some processors (e.g. MIPS) actually has ALU instructions that evaluates some of these operators.)

Program flow control in assembler

- In assembly language
  - Subroutine calls
  - Unconditional jumps
  - Conditional jumps
- Condition test and conditional jump are (often) separated
  - The first instruction computes the flags
  - The second instruction does a conditional jump

Program flow control instructions

Description	Condition	Expression
Jump when less than	<	N=1
Jump when less than or equal to	<=	N=1 or Z=1
Jump when equal to	==	Z=1
Jump when greater than or equal to	>=	N=0
Jump when greater than	>	N=0 and Z=0
Jump when not equal to	!=	Z=0
Unconditional jump	-	-
Jump, push return address	-	-
Set PC from popped return address	-	-

Note: &&, ||, and ! are handled by multiple conditional branches. Flag expression assumes that saturation is used for comparisons!

Target addressing for jumping

- Absolute: 16-bit constant
- Indirect: in a general register
  - Note: required for C programs (function pointers)

Ok, now what?

- We have now specified an instruction set based on C (although some details are missing, see Chapter 5 in the textbook for more information)
- Now we need to check the quality of this instruction set through benchmarking

BDTI Benchmarks

- An example of a widely used benchmark for DSP processors
  - Block transfer: Transfer a data block from one memory to another memory
  - Single FIR: N-tap FIR filter running one data sample
  - Frame FIR: N-tap FIR filter running K data samples
  - IIR: Biquad IIR (2nd order IIR) running one data sample
  - 16-bit division: A positive 16-bit value divided by another positive 16-bit value
  - DCT: 8×8 2D DCT

Benchmark results for Junior

Benchmark	Junior	TSMD
Block transfer of 40 samples:	242	47

Benchmark results for Junior

Benchmark	Junior	TSMD
Block transfer of 40 samples:	242	47
Single sample 16-tap FIR	192	31

Benchmark results for Junior

Benchmark	Junior	TSMD
Block transfer of 40 samples:	242	47
Single sample 16-tap FIR	192	31
40 sample Frame 16-tap FIR	7492	893

Benchmark results for Junior

Benchmark	Junior	TSMD
Block transfer of 40 samples:	242	47
Single sample 16-tap FIR	192	31
40 sample Frame 16-tap FIR	7492	893
…	…	…
	Bad	Good

Study of single sample 16-tap FIR assembler code

- Convolution: $y[n] = \sum_{k=0}^{N-1} h[k]x[n-k]$
- Samples stored in a circular buffer

```
// C code for 16 tap FIR filter (single sample)
result = 0;
for (i=0; i < 16; i++) {
    if(ptr1 == TOP){
        ptr1 = BOTTOM;
    }
    result = result + mem[ptr1++]*mem[ptr2++];
}
```

Study of single sample 16-tap FIR assembler code

- Convolution: $y[n] = \sum_{k=0}^{N-1} h[k]x[n-k]$
- Samples stored in a circular buffer

```
// R0: Pointer into x, R2: Pointer into h, R4: Iteration count
// R5: TOP of circular buffer, R7: Bottom of circular buffer

      CAC   A
Loop  SUB R6,R0,R5        // R6 = R0-R5
      JNE FIFO
      MOVE R0, R7
FIFO  Load R1, DM0(R0++)
      Load R3, DM0(R2++)
      MAC  A,R1,R3
      DEC  R4
      JGT  LOOP
```

Discussion break

- Convolution: $y[n] = \sum_{k=0}^{N-1} h[k]x[n-k]$
- Samples stored in a circular buffer

```
// R0: Pointer into x, R2: Pointer into h, R4: Iteration count
// R5: TOP of circular buffer, R7: Bottom of circular buffer
      CAC A
Loop  SUB R6,R0,R5        // R6 = R0-R5
      JNE FIFO
      MOVE R0, R7
FIFO  Load R1, DM0(R0++)
      Load R3, DM0(R2++)
      MAC  A,R1,R3
      DEC  R4
      JGT  LOOP
```

- *How do we improve this, preferably without too much hardware overhead?*

Improved Instruction Set

- Add a loop instruction, shave off 2 cycles per iteration
  - Alternative: Loop unrolling (although inefficient if unrolled too many times)

```
        CAC A
        REPEAT 16, ENDLOOP
        SUB R6,R0,R5          // R6 = R0-R5
        JNE FIFO
        MOVE R0, R7
FIFO    Load R1, DM0(R0++)
        Load R3, DM0(R2++)
        MAC  A,R1,R3
ENDLOOP
```

Improved Instruction Set

- Add a circular addressing/modulo addressing
  - Alternative: Use longer buffers (inefficient use of memory)

```
        CAC A
        REPEAT 16, ENDLOOP
FIFO    Load R1, DM0(R0++%)
        Load R3, DM0(R2++)
        MAC  A,R1,R3
ENDLOOP
```

Improved Instruction Set

- Let the MAC instruction read operands from the data memory
  - Problem: Dual port memories are expensive

```
        CAC A
        REPEAT 16, ENDLOOP
FIFO    MAC   A,DM0(R0++%), DM0(R2++)
ENDLOOP
```

Improved Instruction Set

- Split DM0 into two data memories, DM0 and DM1

```
      CAC A
      REPEAT 16, ENDLOOP
FIFO        MAC  A,DM0(R0++%), DM1(R2++)
ENDLOOP
```

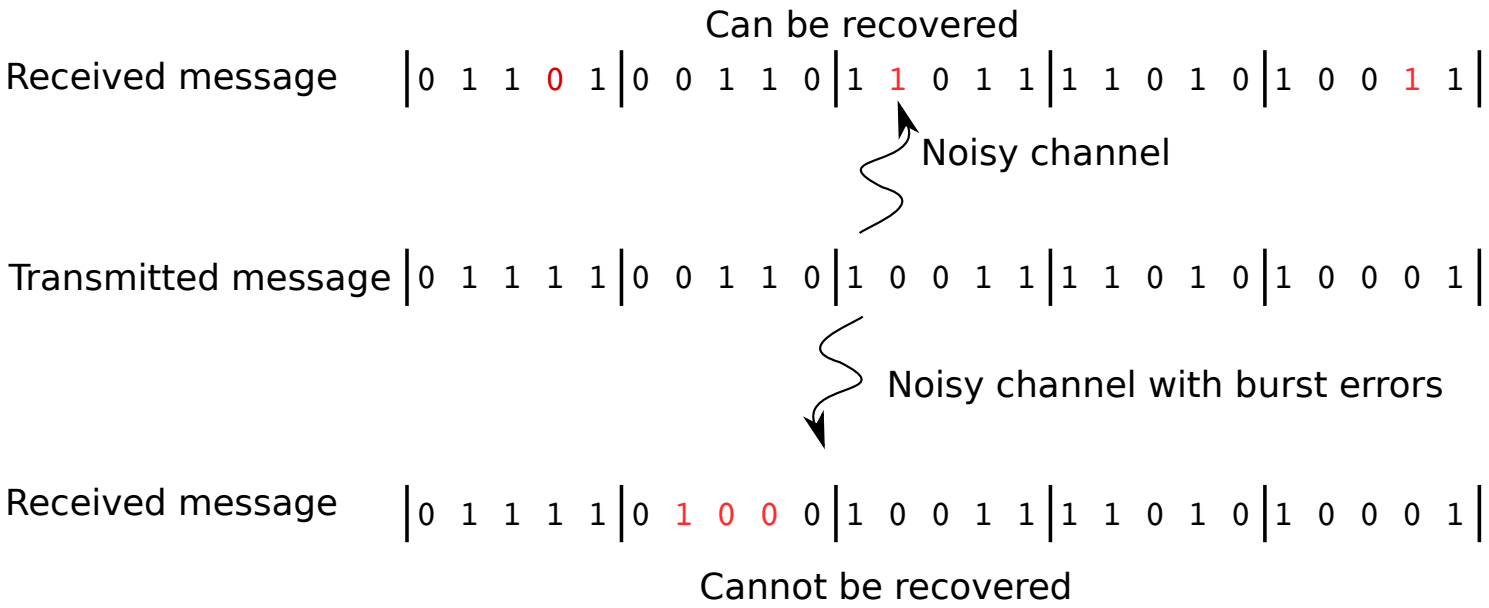
- Final performance: 17 cycles for the kernel + setup costs
- Original performance: About 176 cycles for the kernel + setup costs
- A huge improvement based on fairy small hardware changes aside from splitting the data memory in two

Another example: Interleaving

```
void interleaver(int ptr0, int ptr1)
{
    for(i=0; i < 256; i = i + 1) {
        uint16_t idx = mem[ptr0++]
        uint16_t tmp = mem[idx]
        mem[ptr1++] = tmp
    }
}
```

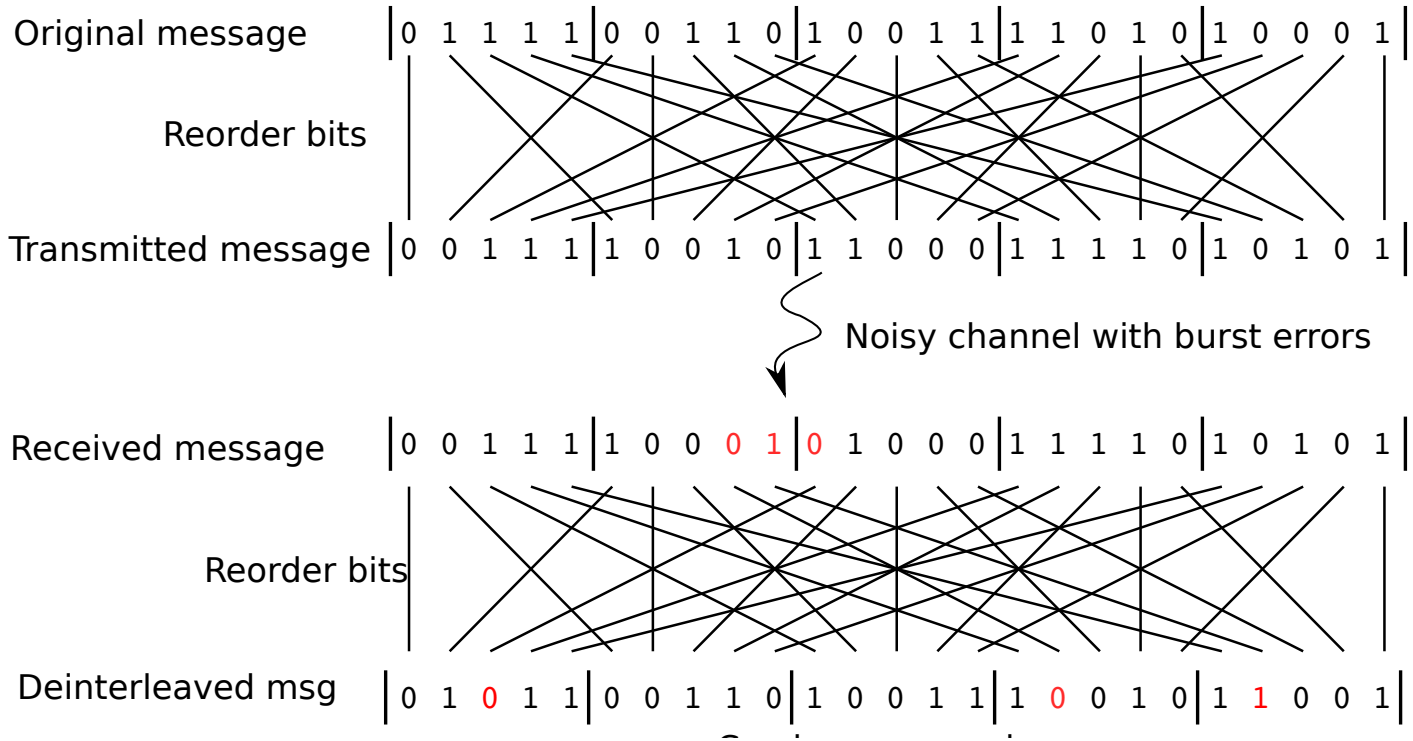
Interleaver? What is that?

- Consider an error correcting code capable of correcting one erroneous bit out of a 5 bit group



Interleaver? What is that?

- By reordering the bits we can improve burst error resilience



Another example: Interleaving

```
void interleaver(int ptr0, int ptr1) {
    for(i=0; i < 256; i = i + 1) {
        uint16_t idx = mem[ptr0++] // ptr0 indexes
        uint16_t tmp = mem[idx]    //  a lookup table
        mem[ptr1++] = tmp
    }
}

// Lookup table could contain (for example)
// 0 16 32 48 64 ... 1 17 33 49 65 ... 2 18 34 50 66, ...
// although other patterns are of course also possible
```

Another example: Interleaving

```
; Assembler code v1:
    repeat 256,endloop
    load  r0,DM0[ptr0++]
    load  r0,DM0[r0]
    store DM0[ptr1++],r0
endloop:

// 768 clock cycles
```

Move some data to DM1

```
; Assembler code v1:
    repeat 256,endloop
    load r0,DM0[ptr0++]
    load r0,DM1[r0]
    store DM0[ptr1++],r0
endloop:

// 768 clock cycles
```

- Discussion break/until next lecture: Think about how this can be improved by modifying the instruction set, addressing modes and/or the architecture of the processor.

Memory indirect addressing

```
; Assembler code v3:
    repeat 256,endloop
    load r0,DM1[DM0[ptr0++]]    ; Combined the loads!
    store DM0[ptr1++],r0
endloop:

// 512 clock cycles
```

Memory indirect addressing

```
; Reality check: Data hazards!
; Assembler code v3:
    repeat 256,endloop
    load r0,DM1[DM0[ptr0++]]
    store DM0[ptr1++],r0
endloop:

// 512 clock cycles
```

- Short discussion break: How to rewrite the code above to avoid the data hazard?

Memory indirect addressing

```
; Assembler code v4:
    repeat 64,endloop
    load r0,DM1[DM0[ptr0++]]  ; Unrolls the loop
    load r1,DM1[DM0[ptr0++]]  ; to avoid data
    load r2,DM1[DM0[ptr0++]]  ; hazards
    load r3,DM1[DM0[ptr0++]]
    store DM0[ptr1++],r0
    store DM0[ptr1++],r1
    store DM0[ptr1++],r2
    store DM0[ptr1++],r3
endloop:
```

Alternative 2: Memory indirect addressing during store

```
; Assembler code v5:
    repeat 256,endloop
    load r0,DM0[ptr0++]
    store DM0[ptr1++],DM1[r0] // DM0 = DM1[r0]
endloop:

// 512 clock cycles
```

- Justification: It is better if the pipeline is created in such a way that a store takes longer time to complete than a load.
  - (A store will seldom generate data dependencies whereas a load to a register will easily generate data dependencies as seen in the first alternative.)

Alternative 2: Memory indirect addressing during store (unrolled)

```
; Assembler code v6:
    repeat 64,endloop
    load r0,DM0[ptr0++]
    load r1,DM0[ptr0++]
    load r2,DM0[ptr0++]
    load r3,DM0[ptr0++]
    // A store buffer can simplify some of the data hazards here.
    // (might still need some unrolling)
    store DM0[ptr1++],DM1[r0]
    store DM0[ptr1++],DM1[r1]
    store DM0[ptr1++],DM1[r2]
    store DM0[ptr1++],DM1[r3]
endloop:

// 512 clock cycles
```

Alternative 3: Rewrite loop as follows

- Output stored in DM1 this time around, remaining data in DM0

```
; Assembler code v7:
    load      r0,DM0[ptr0++]

    repeat 255,endloop
    load      r0,DM0[r0]
    store     DM1[ptr1++],r0
    load      r0,DM0[ptr0++]
endloop
    load r0,DM0[r0]
    store DM1[ptr1++], r0

    // 768 clock cycles for loop, no improvement (yet)
```

Alternative 3: Merge instructions

- No real data dependency between the marked instructions, merge these into one!

```
; Assembler code v7:
    load      r0,DM0[ptr0++]

    repeat 255,endloop
    load      r0,DM0[r0]
    store     DM1[ptr1++],r0 // These two instructions
    load      r0,DM0[ptr0++] // can be merged without
                             // additional HW cost!
endloop
    load r0,DM0[r0]
    store DM1[ptr1++], r0

    // 768 clock cycles for loop, no improvement (yet)
```

Alternative 3: Merge instructions

- A form of software pipelining has been used here
- (The inner loop operates partly on iteration i, and partly on iteration i+1)

```
; Assembler code v8:
    load      r0,DM0[ptr0++]  // Prologue
    repeat 255,endloop
    load      r0,DM0[r0]
    loadstore r0,DM0[ptr0++], DM1[ptr1++],r0 // These two instructions

    // loadstore does the following:
    // DM1[ptr1++] = r0; r0 = DM0[ptr0++]
endloop
    load r0,DM0[r0]           // Epilogue
    store DM1[ptr1++], r0

    // 768 clock cycles for loop, no improvement (yet)
```

Alternative 3: Rewrite loop as follows

- Advantage of alternative 3:
  - The pipeline depth of loadstore is the same as the pipeline depth of load and store
  - The instruction may also be useful in other situations such as when copying values from one memory to another

Conclusions - Instruction set design

- C hides memory addressing costs and loop costs
- At assembly language level, memory addressing must be explicitly executed.
- We can conclude that most memory access and addressing can be pipelined and executed in parallel behind running the arithmetic operations.

Conclusions - Instruction set design

- One essential ASIP design technique will be grouping the arithmetic and memory operations into one specific instruction if they are used together all the time
- Remember this during lab 4!

Conclusions - Instruction set design

- To hide the cost of memory addressing and data access is to design smart addressing models by finding and using regularities of addressing and memory access.
- Addressing regularities:
  - postincremental addressing
  - modulo addressing
  - postincremental with variable step size
  - and bit-reversed addressing.

Oscar Gustafsson

www.liu.se

LINKÖPING UNIVERSITY