# Simulering af digitallogik i 'C' - og A5

# A5 - Simulering af x86prime i 'C'

- Vi giver jer en simulator der ikke er helt færdig
- En simulator er et program som lader som om det er en maskine. Det vil sige: det kan udføre andre programmer skrevet til den maskine der simuleres.
- Senere giver I os en simulator der *er* færdig. Den kan udføre x86prime programmer
- Der er krav til *hvordan* i skriver jeres simulator. De krav afspejler hvordan hardware essentielt virker.

# Modellering i C

Vi har oversat egenskaberne ved byggeklodserne og hvordan de kombineres til *formkrav* til programmer skrevet i C. Det ligner ikke normalt C.

Vi bruger kun to typer i vores program: en til kontrol-signaler, en til data:

- Vi repræsenterer et kontrol signal med en "bool"
- Vi representerer øvrigt data med en "val"

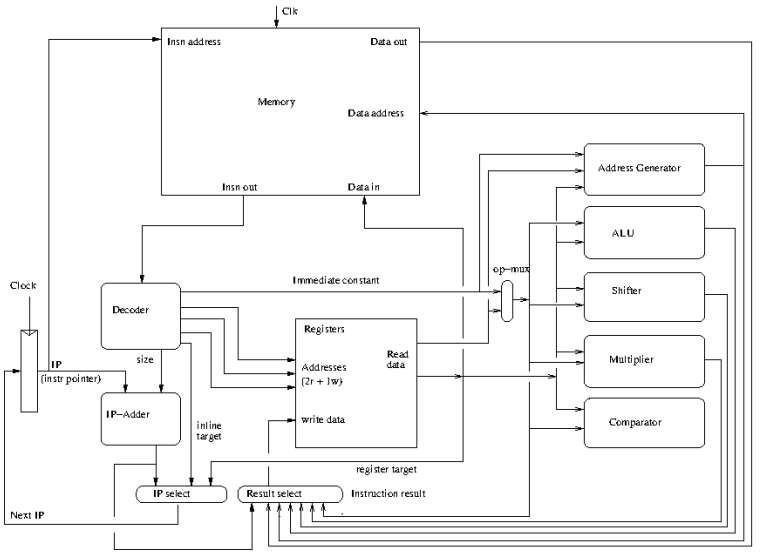
En maskin cyklus udtrykkes ved et løkke gennemløb. Tilstandselementer erklæres udenfor løkken. Alle de funktionelle byggeklodser udfører deres arbejde først i hvert gennemløb. Alle tilstandselementer opdateres i slutningen af hvert gennemløb.

```
...initialisering...
while (..) {
  // funktionelle byggeklodser:
  ...tildelinger og funktionskald...

  // clock-puls... opdatering af tilstandselementer:
  ...opdatering af registre, herunder IP/PC...
}
```

# Lad os kigge lidt på den udleverede simulator

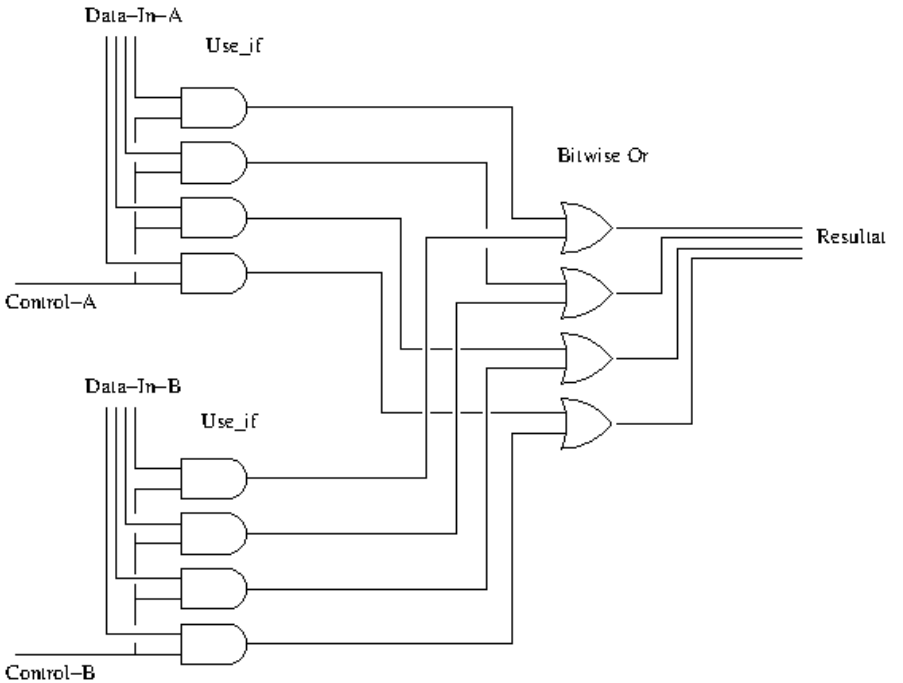
# Spot sammenhængen




```
val agen_result = address_generate(reg_out_a, reg_out_b, sext_imm_i,
                  shamt, use_z, use_s, use_d);
val alu_result = alu_execute(alu_ctrl, reg_out_a, op_b);
val mul_result = multiplier(mul_is_signed, reg_out_a, op_b);
val shifter_result = shifter(shft_is_left, shft_is_signed, reg_out_a, op_b);
val compute_result = or(use_if(use_agen, agen_result),
           or(use_if(use_multiplier, mul_result),
             or(use_if(use_shifter, shifter_result),
               or(use_if(use_direct, op_b), use_if(use_alu, alu_result)))));
```

# Valg mellem mulige resultater

Vi har ikke if..then..else.. vi bruger en multiplexor



# Udleveret kode - Forbindelser

```
// simple conversion
val from_int(uint64_t);

// pick a set of bits from a value
val pick_bits(int lsb, int sz, val);

// pick a single bit from a value
bool pick_one(int position, val);

// sign extend by copying a sign bit to all higher positions
val sign_extend(int sign_position, val value);
```

Og flere, men ovenstående er de mest betydende

# Udleveret kode - Logiske operationer

For kontrol signaler, repræsenteret ved bool's bruger man bare de indbyggede logiske operatorer.

For data, repræsenteret ved typen val findes der følgende funktioner

```
// mask out a value if control is false
val use_if(bool control, val value);

// bitwise and, or, xor and negate for bitvectors
val and(val a, val b);
val or(val a, val b);
val xor(val a, val b);
val neg(int num_bits, val);

// reduce a bit vector to a bool by and'ing or or'ing all elements
bool reduce_and(int num_bits, val);
bool reduce_or(val);

// 64 bit addition
val add(val a, val b);
```

Vi udleverer også alle beregnings-enheder (alu, shifter, etc).

# Et Kig på den udleverede kode

Lad os gennemgå "main.c"

# Brug af den ufærdige simulator

Først skal vi have et program i 'prime' assembler. Vi bruger en gammel kending:

```
00000000 :                    #  Start:
00000000 : A45030000000         #      leaq data, %rsi
00000006 : 640000000000         #      movq $0, %rax
0000000c : 644000000000         #      movq $0, %rbp
00000012 : 93B503               #      leaq (%rsi, %rax, 8), %r11
00000015 : 313B                 #      movq (%r11), %rdx
00000017 : 500001000000         #      addq $1, %rax
0000001d : 1043                 #      addq %rdx, %rbp
0000001f : 93B503               #      leaq (%rsi, %rax, 8), %r11
00000022 : 313B                 #      movq (%r11), %rdx
00000024 : 500001000000         #      addq $1, %rax
0000002a : 1043                 #      addq %rdx, %rbp
0000002c : 0000                 #      stop
00000030 :                    #      .align 8
00000030 :                    #  data:
00000030 : 2a00000000000000     #      .quad 42
00000038 : 1500000000000000     #      .quad 21
```

Det er en stor hjælp :-)

# Prun (Prime Run)

Prun kører (simulerer) et program i hex format:

```
./prun fragment.hex Start -show
Starting execution from address 0x0

00000000 : a4 50 00000030          leaq 0x30, %rsi          %rsi <- 0x30
00000006 : 64 00 00000000          movq $0, %rax            %rax <- 0x0
0000000c : 64 40 00000000          movq $0, %rbp            %rbp <- 0x0
00000012 : 93 b5 03              leaq (%rsi, %rax, 8), %r11   %r11 <- 0x30
00000015 : 31 3b                 movq (%r11), %rdx        %rdx <- 0x2a
00000017 : 50 00 00000001          addq $1, %rax            %rax <- 0x1
0000001d : 10 43                 addq %rdx, %rbp          %rbp <- 0x2a
0000001f : 93 b5 03              leaq (%rsi, %rax, 8), %r11   %r11 <- 0x38
00000022 : 31 3b                 movq (%r11), %rdx        %rdx <- 0x15
00000024 : 50 00 00000001          addq $1, %rax            %rax <- 0x2
0000002a : 10 43                 addq %rdx, %rbp          %rbp <- 0x3f
0000002c : 00 00                 stop
Terminated by STOP instruction
```

# Den udleverede simulator

Den udleverede simulator skal have udpeget en "hex" fil (lavet med 'prasm') og en start adresse. Den er jo ufærdig, så hvad mon der sker, hvis man forsøger at køre et program?

```
> ./sim fragment.hex 0
1 0
2 2
3 4
Done
```

Udskriften siger ikke meget. Den kommer fra toppen af løkken i main.c og angiver blot instruktionsnummeret og adressen på instruktionen. Man kan se at adresserne 0, 2, 4 ikke matcher placeringen af de første instruktioner og at afviklingen stopper før tid.

Jævnfør udskriften fra kørslen med reference simulatoren, forrige slide.

# Fejlfinding

'prun' (også kaldet reference-simulatoren) kan producere en sporings-fil - en fil med de sideeffekter et program har mens det udføres.

Den udleverede simulator til a5 (den I skal færddiggøre) kan læse sådan en sporings-fil og sammenholde den med hvad der sker under simulationen.

Hvis der detekteres en afvigelse fra "sporet", kaldes en funktion der hedder error() med en fejlmeddelelse. Meddelelsen skrives ud og programmet terminerer.

Men man kan køre simulatoren i gdb og sætte et breakpoint på error(). Derefter er det ligetil at inspicere variable i programmet

Lad os prøve det!

# Fejlfinding (II)

Produktion af en tracefile:

```
> ./prun fragment.hex Start -tracefile fragment.trc
> less fragment.trc
P 0 0
R 5 30
P 0 6
R 0 0
P 0 c
R 4 0
P 0 12
R b 30
P 0 15
R 3 2a
```

Hver linie i tracefilen angiver en opdatering af maskinens tilstand. Linierne med 'P' er opdateringer til instruktionspegeren, 'R' er opdatering af et register og 'M' (ikke vist her) er opdateringer til et ord i lageret. I behøver ikke kunne læse en tracefile - men den udleverede simulator kan og vil holde simulatorens opførsel op i mod tracefilen.

# Fejlfinding (III)

Vi kan angive en tracefile som det tredje argument:

```
./sim fragment.hex 0 fragment.trc
1 0
 -- value mismatch, access 'P' 0
 -- with value 2, but tracefile expected 6
Trace mismatch on write to instruction pointer
```

Simulationen stoppes nu allerede i slutningen af første clock-cycle (løkke-gennemløb) med en besked (lidt kryptisk) om at IP forsøges opdateret med værdien '2', men det skulle have været '6'. Baggrund er selvfølgelig at den første instruktion fylder 6 bytes, så den næste IP skal være 6. Men da den udleverede simulator bare antager at alle instruktioner fylder to bytes, så er dens opdatering af IP forkert.

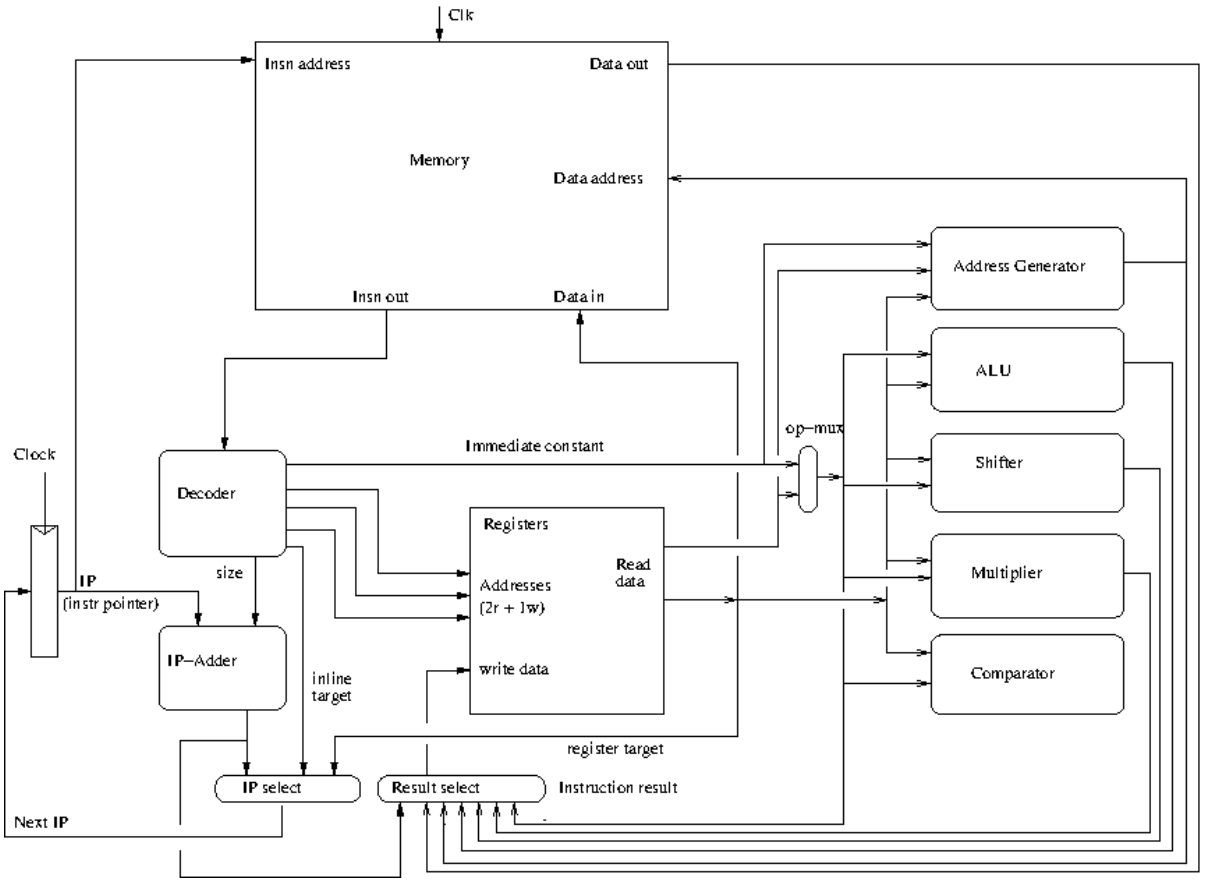
# Fejlfinding (IV)

Med 'gdb' kan man få adgang til alle værdierne i hovedløkken og dermed tjekke om man beregner sine signaler korrekt. For at gøre det skal man sætte et breakpoint i funktionen 'error':

```
> gdb sim
...
(gdb) set args fragment.hex 0 fragment.trc
(gdb) br error
Breakpoint 1 at 0x555555557dce: error. (2 locations)
(gdb) r
Starting program: /home/finn/ark/A5/src/sim fragment.hex 0 fragment.trc
1 0
 -- value mismatch, access 'P' 0
 -- with value 2, but tracefile expected 6

Breakpoint 1, error (message=0x55555555ddd0 "\360\333UUUU") at support.c:6
6    void error(const char* message) {
(gdb) bt
#0  error (message=0x55555555ddd0 "\360\333UUUU") at support.c:6
#1  0x0000555555556055 in ip_write (ip_reg=0x55555555d2a0, value=..., wr_enable=true) at ip_reg.c:46
#2  0x0000555555556faf in main (argc=4, argv=0x7fffffffde78) at main.c:238
(gdb) frame 2
#2  0x0000555555556faf in main (argc=4, argv=0x7fffffffde78) at main.c:238
238             ip_write(ip, pc_next, true);
(gdb) print pc_next
```

# A5 mikroarkitektur




Spørgsmål? Klart som blæk?

# Spørgsmål og Svar