ARC

Machine architecture

Assembly

Registers

The "ra" register holds the return address

call func, stores pc + 4 in ra.

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

Registers

X1 holds the address which will be jumped to when running *ret*

X10-17 contains function arguments. Caller writes to these registers, and callee reads from them.

X2/sp can be descreased to hold additional arguments on the stack.

Instructions

Category	Name	Fmt	RV32I Base	
Loads	Load Byte	I	`LB`	`rd,rs1,imm`
	Load Halfword	I	`LH`	`rd,rs1,imm`
	Load Word	I	`LW`	`rd,rs1,imm`
	Load Byte Unsigned	I	`LBU`	`rd,rs1,imm`
	Load Half Unsigned	I	`LHU`	`rd,rs1,imm`
Stores	Store Byte	S	`SB`	`rs1,rs2,imm`
	Store Halfword	S	`SH`	`rs1,rs2,imm`
	Store Word	S	`SW`	`rs1,rs2,imm`
Shifts	Shift Left	R	`SLL`	`rd,rs1,rs2`
	Shift Left Immediate	I	`SLLI`	`rd,rs1,shamt`
	Shift Right	R	`SRL`	`rd,rs1,rs2`
	Shift Right Immediate	I	`SRLI`	`rd,rs1,shamt`
	Shift Right Arithmetic	R	`SRA`	`rd,rs1,rs2`
	Shift Right Arith Imm	I	`SRAI`	`rd,rs1,shamt`
Arithmetic	ADD	R	`ADD`	`rd,rs1,rs2`
	ADD Immediate	I	`ADDI`	`rd,rs1,imm`
	SUBtract	R	`SUB`	`rd,rs1,rs2`
	Load Upper Imm	U	`LUI`	`rd,imm`
	Add Upper Imm to PC	U	`AUIPC`	`rd,imm`
Logical	XOR	R	`XOR`	`rd,rs1,rs2`
	XOR Immediate	I	`XORI`	`rd,rs1,imm`
	OR	R	`OR`	`rd,rs1,rs2`
	OR Immediate	I	`ORI`	`rd,rs1,imm`
	AND	R	`AND`	`rd,rs1,rs2`
	AND Immediate	I	`ANDI`	`rd,rs1,imm`
Compare	Set <	R	`SLT`	`rd,rs1,rs2`
	Set < Immediate	I	`SLTI`	`rd,rs1,imm`
	Set < Unsigned	R	`SLTU`	`rd,rs1,rs2`
	Set < Imm Unsigned	I	`SLTIU`	`rd,rs1,imm`
Branches	Branch =	SB	`BEQ`	`rs1,rs2,imm`
	Branch ≠	SB	`BNE`	`rs1,rs2,imm`
	Branch <	SB	`BLT`	`rs1,rs2,imm`
	Branch ≥	SB	`BGE`	`rs1,rs2,imm`

Branch ≥	SB	BGE	rs1,rs2,imm
Branch < Unsigned	SB	BLTU	rs1,rs2,imm
Branch ≥ Unsigned	SB	BGEU	rs1,rs2,imm
Jump & Link J&L	UJ	JAL	rd,imm
Jump & Link Register	UJ	JALR	rd,rs1,imm

Category	*Name*	*Fmt*	*RV32M (Multiply-Divide)*	
Multiply	MULtiply	R	MUL	rd,rs1,rs2
	MULtiply upper Half	R	MULH	rd,rs1,rs2
	MULtiply Half Sign/Uns	R	MULHSU	rd,rs1,rs2
	MULtiply upper Half Uns	R	MULHU	rd,rs1,rs2
Divide	DIVide	R	DIV	rd,rs1,rs2
	DIVide Unsigned	R	DIVU	rd,rs1,rs2
Remainder	REMainder	R	REM	rd,rs1,rs2
	REMainder Unsigned	R	REMU	rd,rs1,rs2

Arithmetic Operation

Mnemonic	Instruction	Type	Description
ADD rd, rs1, rs2	Add	R	rd ← rs1 + rs2
SUB rd, rs1, rs2	Subtract	R	rd ← rs1 - rs2
ADDI rd, rs1, imm12	Add immediate	I	rd ← rs1 + imm12
SLT rd, rs1, rs2	Set less than	R	rd ← rs1 < rs2 ? 1 : 0
SLTI rd, rs1, imm12	Set less than immediate	I	rd ← rs1 < imm12 ? 1 : 0
SLTU rd, rs1, rs2	Set less than unsigned	R	rd ← rs1 < rs2 ? 1 : 0
SLTIU rd, rs1, imm12	Set less than immediate unsigned	I	rd ← rs1 < imm12 ? 1 : 0
LUI rd, imm20	Load upper immediate	U	rd ← imm20 << 12
AUIP rd, imm20	Add upper immediate to PC	U	rd ← PC + imm20 << 12

Logical Operations

Mnemonic	Instruction	Type	Description
`AND rd, rs1, rs2`	AND	R	`rd ← rs1 & rs2`
`OR rd, rs1, rs2`	OR	R	`rd ← rs1 | rs2`
`XOR rd, rs1, rs2`	XOR	R	`rd ← rs1 ^ rs2`
`ANDI rd, rs1, imm12`	AND immediate	I	`rd ← rs1 & imm12`
`ORI rd, rs1, imm12`	OR immediate	I	`rd ← rs1 | imm12`
`XORI rd, rs1, imm12`	XOR immediate	I	`rd ← rs1 ^ imm12`
`SLL rd, rs1, rs2`	Shift left logical	R	`rd ← rs1 << rs2`
`SRL rd, rs1, rs2`	Shift right logical	R	`rd ← rs1 >> rs2`
`SRA rd, rs1, rs2`	Shift right arithmetic	R	`rd ← rs1 >> rs2`
`SLLI rd, rs1, shamt`	Shift left logical immediate	I	`rd ← rs1 << shamt`
`SRLI rd, rs1, shamt`	Shift right logical imm.	I	`rd ← rs1 >> shamt`
`SRAI rd, rs1, shamt`	Shift right arithmetic immediate	I	`rd ← rs1 >> shamt`

Mnemonic	Instruction	Type	Description
`LD rd, imm12(rs1)`	Load doubleword	I	`rd ← mem[rs1 + imm12]`
`LW rd, imm12(rs1)`	Load word	I	`rd ← mem[rs1 + imm12]`
`LH rd, imm12(rs1)`	Load halfword	I	`rd ← mem[rs1 + imm12]`
`LB rd, imm12(rs1)`	Load byte	I	`rd ← mem[rs1 + imm12]`
`LWU rd, imm12(rs1)`	Load word unsigned	I	`rd ← mem[rs1 + imm12]`
`LHU rd, imm12(rs1)`	Load halfword unsigned	I	`rd ← mem[rs1 + imm12]`
`LBU rd, imm12(rs1)`	Load byte unsigned	I	`rd ← mem[rs1 + imm12]`
`SD rs2, imm12(rs1)`	Store doubleword	S	`rs2 → mem[rs1 + imm12]`
`SW rs2, imm12(rs1)`	Store word	S	`rs2(31:0) → mem[rs1 + imm12]`
`SH rs2, imm12(rs1)`	Store halfword	S	`rs2(15:0) → mem[rs1 + imm12]`
`SB rs2, imm12(rs1)`	Store byte	S	`rs2(7:0) → em[rs1 + imm12]`

Branching

Mnemonic	Instruction	Type	Description
`BEQ rs1, rs2, imm12`	Branch equal	SB	`if rs1 == rs2` `pc ← pc + imm12`
`BNE rs1, rs2, imm12`	Branch not equal	SB	`if rs1 != rs2` `pc ← pc + imm12`
`BGE rs1, rs2, imm12`	Branch greater than or equal	SB	`if rs1 >= rs2` `pc ← pc + imm12`
`BGEU rs1, rs2, imm12`	Branch greater than or equal unsigned	SB	`if rs1 >= rs2` `pc ← pc + imm12`
`BLT rs1, rs2, imm12`	Branch less than	SB	`if rs1 < rs2` `pc ← pc + imm12`
`BLTU rs1, rs2, imm12`	Branch less than unsigned	SB	`if rs1 < rs2` `pc ← pc + imm12 << 1`
`JAL rd, imm20`	Jump and link	UJ	`rd ← pc + 4` `pc ← pc + imm20`
`JALR rd, imm12(rs1)`	Jump and link register	I	`rd ← pc + 4` `pc ← rs1 + imm12`

Pseudo instructions

A pseudo instruction is an instruction handled by
the assembler by translating it into one or more
real (non-pseudo) instructions. mv will use addi to move a value from one register to another.

Pseudoinstruction	Base Instruction(s)	Meaning
`la rd, symbol`	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	Load address
`l{b|h|w|d} rd, symbol`	`auipc rd, symbol[31:12]` `l{b|h|w|d} rd, symbol[11:0](rd)`	Load global
`s{b|h|w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `s{b|h|w|d} rd, symbol[11:0](rt)`	Store global
`fl{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fl{w|d} rd, symbol[11:0](rt)`	Floating-point load global
`fs{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fs{w|d} rd, symbol[11:0](rt)`	Floating-point store global
`nop`	`addi x0, x0, 0`	No operation
`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if = zero
`snez rd, rs`	`sltu rd, x0, rs`	Set if ≠ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if < zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if > zero
`fmv.s rd, rs`	`fsgnj.s rd, rs, rs`	Copy single-precision register
`fabs.s rd, rs`	`fsgnjx.s rd, rs, rs`	Single-precision absolute value
`fneg.s rd, rs`	`fsgnjn.s rd, rs, rs`	Single-precision negate
`fmv.d rd, rs`	`fsgnj.d rd, rs, rs`	Copy double-precision register
`fabs.d rd, rs`	`fsgnjx.d rd, rs, rs`	Double-precision absolute value
`fneg.d rd, rs`	`fsgnjn.d rd, rs, rs`	Double-precision negate
`beqz rs, offset`	`beq rs, x0, offset`	Branch if = zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if ≠ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if ≤ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if ≥ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if < zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if > zero
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if >
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if ≤
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if >, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if ≤, unsigned
`j offset`	`jal x0, offset`	Jump
`jal offset`	`jal x1, offset`	Jump and link
`jr rs`	`jalr x0, rs, 0`	Jump register
`jalr rs`	`jalr x1, rs, 0`	Jump and link register
`ret`	`jalr x0, x1, 0`	Return from subroutine
`call offset`	`auipc x6, offset[31:12]` `jalr x1, x6, offset[11:0]`	Call far-away subroutine
`tail offset`	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`	Tail call far-away subroutine
`fence`	`fence iorw, iorw`	Fence on all memory and I/O

Mnemonic	Instruction	Base instruction(s)
`LI rd, imm12`	Load immediate (near)	`ADDI rd, zero, imm12`
`LI rd, imm`	Load immediate (far)	`LUI rd, imm[31:12]` `ADDI rd, rd, imm[11:0]`
`LA rd, sym`	Load address (far)	`AUIPC rd, sym[31:12]` `ADDI rd, rd, sym[11:0]`
`MV rd, rs`	Copy register	`ADDI rd, rs, 0`
`NOT rd, rs`	One's complement	`XORI rd, rs, -1`
`NEG rd, rs`	Two's complement	`SUB rd, zero, rs`
`BGT rs1, rs2, offset`	Branch if rs1 > rs2	`BLT rs2, rs1, offset`
`BLE rs1, rs2, offset`	Branch if rs1 ≤ rs2	`BGE rs2, rs1, offset`
`BGTU rs1, rs2, offset`	Branch if rs1 > rs2 (unsigned)	`BLTU rs2, rs1, offset`
`BLEU rs1, rs2, offset`	Branch if rs1 ≤ rs2 (unsigned)	`BGEU rs2, rs1, offset`
`BEQZ rs1, offset`	Branch if rs1 = 0	`BEQ rs1, zero, offset`
`BNEZ rs1, offset`	Branch if rs1 ≠ 0	`BNE rs1, zero, offset`
`BGEZ rs1, offset`	Branch if rs1 ≥ 0	`BGE rs1, zero, offset`
`BLEZ rs1, offset`	Branch if rs1 ≤ 0	`BGE zero, rs1, offset`
`BGTZ rs1, offset`	Branch if rs1 > 0	`BLT zero, rs1, offset`
`J offset`	Unconditional jump	`JAL zero, offset`
`CALL offset12`	Call subroutine (near)	`JALR ra, ra, offset12`
`CALL offset`	Call subroutine (far)	`AUIPC ra, offset[31:12]` `JALR ra, ra, offset[11:0]`
`RET`	Return from subroutine	`JALR zero, 0(ra)`

NOP	No operation	ADDI zero, zero, 0

Calling functions

Look for Jumping instructions to determine if a function is being called (jal, jalr). is function argument registers being used (x10-x17 or a0-a7)

Function arguments can be stored and loaded from the stack.

Argument registers are recognized by using the register in a function without first having written to it in that function.

OBS: jalr x0, x1, 0 does NOT call functions it returns from a function. (x1 contains the return address)
• Functions can be called with the following pseudo instructions:
• j offset :jump
• jal offset :jump and link (return addr is stored in x1)
• jr rs :jump register
• jalr rs :jump and link register (return addr is stored in x1)
• call offset :jump and link to far away address (return addr is stored in x1)

OBS: the pseudo instruction ret (instruction jalr x0, x1, 0) does NOT call other functions, it returns from a function. (can also be written jalr x0, ra, 0)

Example of function

• Caller loads arguments into registers a0-a7.
• Caller calls function (jal, jalr, etc.)
• Callee uses arguments
• Callee stores result (if any) in a0-1
• Callee returns (ret)
• Caller uses results (if any) from a0-1

Loops

Branching instructions or pseudo branch instructions will often be used in a loop.

While-loop, For-loop optionally have a jalr or jal instruction (j pseudo instruction)

OBS: Infinite while-loop can use pseudo instruction j offset (instruction jal x0, offset , negative offset) with no branch instruction.

Translating from Assembly to C and general tip

Risc v uses goto style, C does not.

Take the assembly code and write a comment for each line that in c like pseudo code and explain with words what happens in that line

lbu a5, 0(a0) would be a5 = a0

```
// myfunc:
//     lbu a5,0(a0)            // unsigned char* a5 = *a0 (a0 is the address of a first byte/char))
//     beq a5,zero,.L2         // if a5 != 0 -> loop
// .L3:
//     sb a5,0(a1)             // *a1 = a5 (a1 is the address of ANOTHER first byte/char)
//     lbu a5,1(a0)            // unsigned char* a5 = *(a0+1) (a0 is the address of a first byte/char))
//     addi a0,a0,1            // a0 ++
//     addi a1,a1,1            // a1 ++
//     bne a5,zero,.L3         // repeat check beq. if a5 != 0 -> loop
// .L2:
//     sb zero,0(a1)           // *a1 = 0;
//     ret                      // return to addr x1/ra

// a0 = from, a1 = to
char* myfunc_correct(char* a0, char* a1) {
    unsigned char a5 = *a0;
    while (a5 != 0) {
        *a1 = a5;
        a5 = *(a0+1);
        a0 ++;
        a1 ++;
    }
    *a1 = 0;
    return a0;
}
```

Pipeline Teori

The non-pipelined approach to laundry would be as follows:

1. Place one dirty load of clothes in the washer.
2. When the washer is finished, place the wet load in the dryer.

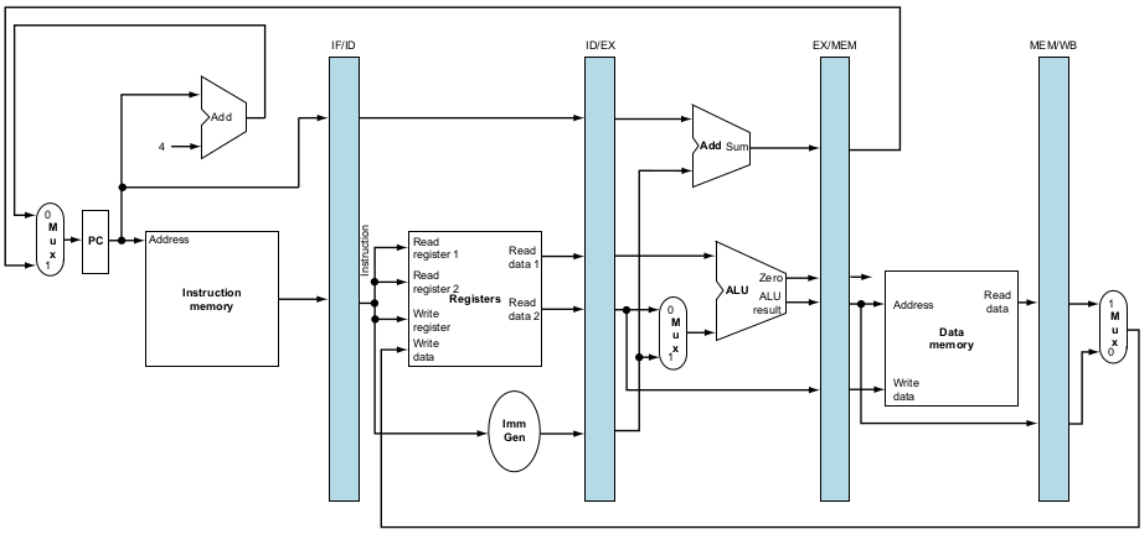
3. When the dryer is finished, place the dry load on a table and fold.
4. When folding is finished, ask your roommate to put the clothes away.

   As soon as the washer is finished with the first load and placed in the dryer, you load the washer with the second dirty load. When the first load is dry, you place it on the table to start folding, move the wet load to the dryer, and put the next dirty load into the washer. Next, you have your roommate put the first load away, you start folding the second load, the dryer has the third load, and you put the fourth load into the washer. At this point all steps—called stages in pipelining—are operating concurrently.

instructions run in parrallel, since more instructions are handled at a time the program is executed faster

The same principles apply to processors where we pipeline instruction execution. RISC-V instructions classically take five steps:

1. FE: Fetch instruction from memory.
2. DE: Read registers and decode the instruction.
3. EX: Execute the operation or calculate an address.
4. ME: Access an operand in data memory (if necessary)
5. WB: Write the result into a register (if necessary).



The pipeline registers, in color, separate each pipeline stage.

In our case each stage takes one clock cycle to complete, when it doesn't get stalled

Pipeline Hazards

There are situations in pipelining when the next instruction cannot execute in the following clock cycle (tal på x akse i afviklingsplot). These events are called hazards, and there are three different types.

Structural hazard: When a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute.

Data hazard(stall): Also called a pipeline data hazard. When a planned instruction cannot execute in the proper clock cycle because data that are needed to execute the instruction are not yet available. For example, suppose we have an add instruction
followed immediately by a subtract instruction that uses that sum (x19):

```
add *x19*, x0, x1
sub x2, *x19*, x3
```

The primary solution is based on the observation that we don’t need to wait for the instruction to complete before trying to resolve the data hazard. For the code sequence above, as soon as the ALU creates the sum for the add, we can supply it as an input for the subtract. Adding extra hardware to retrieve the missing item early from the internal resources is called forwarding or
bypassing.

Info

Officially called a **pipeline stall**, but often given the nickname **bubble**. A stall initiated
in order to resolve a
hazard.

Ekstra

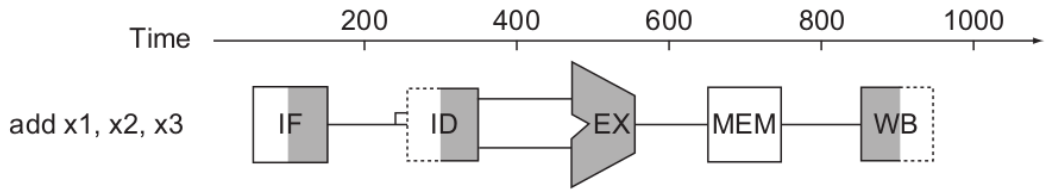


FIGURE 4.30 Graphical representation of the instruction pipeline, similar in spirit to the laundry pipeline in Figure 4.27. Here we use symbols representing the physical resources with the abbreviations for pipeline stages used throughout the chapter. The symbols for the five stages: *IF* for the instruction fetch stage, with the box representing instruction memory; *ID* for the instruction decode/register file read stage, with the drawing showing the register file being read; *EX* for the execution stage, with the drawing representing the ALU; *MEM* for the memory access stage, with the box representing data memory; and *WB* for the write-back stage, with the drawing showing the register file being written. The shading indicates the element is used by the instruction. Hence, MEM has a white background because `add` does not access the data memory. Shading on the right half of the register file or memory means the element is read in that stage, and shading of the left half means it is written in that stage. Hence the right half of ID is shaded in the second stage because the register file is read, and the left half of WB is shaded in the fifth stage because the register file is written.

![[Pasted image 20240115034327.png]

Summary

Pipelining is a technique that exploits parallelism between the instructions in a sequential instruction stream. Pipelining increases the number of simultaneously executing instructions and the rate at which instructions are started and completed. Pipelining does not reduce the time it takes to complete an individual instruction,
also called the latency. For example, the five-stage pipeline still takes five clock cycles for the instruction to complete. pipelining improves instruction throughput rather than individual
instruction execution time or latency.

Stalls and fowarding

Instruction	Execution/address calculation stage control lines		Memory access stage control lines			Write-back stage control lines	
	ALUOp	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	10	0	0	0	0	1	0
lw	00	1	0	1	0	1	1
sw	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X

FIGURE 4.51 The values of the control lines are the same as in Figure 4.22, but they have been shuffled into three groups corresponding to the last three pipeline stages.

Let's look at a sequence with many dependences, shown in color:

```
sub   x2, x1, x3    // Register z2 written by sub
and   x12, x2, x5   // 1st operand(x2) depends on sub
or    x13, x6, x2   // 2nd operand(x2) depends on sub
add   x14, x2, x2   // 1st(x2) & 2nd(x2) depend on sub
sw    x15, 100(x2)  // Base (x2) depends on sub
```

The last four instructions are all dependent on the result in register x2 of the first instruction. If register x2 had the value 10 before the subtract instruction and −20 afterwards, the programmer intends that −20 will be used in the following instructions that refer to register x2.

Pipeline praktisk (Finns noter)

1. FE: Fetch instruction from memory.
2. DE: Read registers and decode the instruction.
3. EX: Execute the operation or calculate an address.
4. ME: Access an operand in data memory (if necessary)
5. WB: Write the result into a register (if necessary).

Full fowarding

Vi antager

- At hop forudsiges taget i De.
- At load data kan forwardes fra starten af Wb til store i Me
  Vi kan beskrive maskinens ressourcer ved antallet af instruktioner

som kan udføre samme aktivitet på samme tidspunkt. For den simple pipeline angives ressourcebegrænsningen ved

```
Fe: 1, De: 1, Ex: 1, Me: 1, Wb: 1
```

kun en instruktion kan være i hver fase samtidig.

">>" indikerer at den stalle i den fase som efterfølger, så "`De >> Ex`", betyder et stall i Ex. Man kan også skrive `De De Ex`, fordi den holder i `De` fasen fordi `Ex` ikke er tilgænglig. Jeg synes den sidste metode er mere overskuelig fordi det bliver nemmere at spotte hvornår der skal ske stalls pga. ressourcebegrænsning

Dataafhængigheder

Dataafhængigheder specificeres ved at angive hvilke aktiviteter der producerer og/eller afhænger af en værdi. Eksempel:

```
load: "Fe De Ex Me Wb"   depend(Ex,rs1), produce(Me,rd)
store: "Fe De Ex Me"     depend(Ex,rs1), depend(Me,rs2)
andre: "Fe De Ex Wb"     depend(Ex,rs1), depend(Ex,rs2),
produce(Ex,rd)
branch: "Fe De Ex" // Wb behøves med betinget hop
```

Her refererer "rs1", "rs2" og "rd" til de to kilderegistre og destinationsregisteret. Ideen er at en instruktion der anfører depend(Ex,rs1) tidligst kan gennemføre "Ex" i en cyklus efter at rs1 er blevet produceret.

Load instruktioner producerer f.eks. deres register i `Me` fasen og derfor må en add instruktion f.eks vente til at load er færdig med sin `Me` før add kan begynde sin `Ex` fase

• **Format på instruktioner:**
• Load: rd, imm(rs1)
• Store: rs2, imm(rs1)
• Alle andre: rd, rs1, rs2/imm/label
• **Eksempler på dataafhængigheder**
• depend(Ex, rs1) betyder at rs1 skal være klar inden denne instruktion kan udføre Ex.
• produce(Wb, rd) betyder at rd bliver skrevet til registeret i Wb-fasen af denne instruktion.
• depend(Fe, PC) betyder at program counter skal være klar før denne instruktion kan gå ind i Fe (Fetch-fasen).

Kontrolafhængigheder

Kontrolafhængigheder specificeres på samme måde som dataafhængigheder men med angivelse af et særlig register: "PC". Eksempel:

```
retur: produce(Ex, PC)
alle: depend(Fe, PC)
```

Angiver at PC opdateres i "Ex" af retur instruktionen og at efter en retur-instruktion kan maskinen tidligst gennemføre "Fe" (instruktionshentning) for efterfølgende instruktioner, når PC er opdateret. Den sidste regel for alle instruktioner: "depend(Fe, PC)" er så indlysende at vi ikke vil anføre den fremover.

Gennemgået eksempel

```
                        0  1  2  3  4  5  6  7  8
lw   x11,0(x10)         Fe De Ex Me Wb
addi x11,x11,100           Fe De De Ex Me Wb
sw   x11,0(x14)               Fe Fe De Ex Me Wb
addi x10,x10,1                   Fe De Ex Me Wb
```

Her er skal addi bruge x11 og er derfor nødt til at vente på at `lw` har hentet en værdi fra memory og lagt den i x11, det sker i `Me`: derfor må addi stalle `Ex` til at `lw` er færdi med `Me`.

I `sw` ser vi at `De` er nødt til at blive stallet fordi vi kun har en ressource til at køre `De` skridtet og derfor sker der et stall på `De` (`Fe` `Fe` eller `Fe >>`) da vi venter på at `addi` er færdig med at stalle. Derfor behøver vi heller ikke stalle `Ex` hos `sw` da vi pga tidligere stall nu har x11 klar fra addi.

addi x10 bliver også udskudt fordi vi ikke kan fetche flere ting på en gang.

Branching

```
hop baglæns taget:       produce(De, PC)
hop baglæns ikke taget:  produce(Ex, PC)
hop forlæns taget:       produce(Ex, PC)
hop forlæns ikke taget:  -
Betinget hop behøver ikke Wb
```

Her er en sekvens af instruktioner med to hop bagud, det første tages, det andet ikke:

```
                          0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
0:      addi    x11,x11,4 Fe De Ex Wb
4:      lw      x12,0(x11)   Fe De Ex Me Wb
8:      add     x13,x13,x12     Fe >> De Ex Wb
C:      bne     x11,x15,0          >> Fe De Ex           <-- NB: Betingede hop behøver ikke
0:      addi    x11,x11,4                     Fe De Ex Wb
4:      lw      x12,0(x11)                       Fe De Ex Me Wb
8:      add     x13,x13,x12                         Fe >> De Ex Wb
C:      bne     x11,x15,0                              >> Fe De Ex
10:     et-eller-andet                                           Fe De Ex ...
```

Det betyder at hvis et hop baglæns tages så vil PC være klar i De derfor kan næste instruktions Fe fase begynde sammtidig med Ex fasen for branch instruktionen. Hvis hop ikke tages eller et hop forlæns tages så kan næste instruktions Fe fase først

her unlades `Me` fasen i add og branch instruktioner: Don't do that.

Superskalar

En maskine der kan udføre to eller flere instruktioner samtidigt kaldes "superskalar".

2 way - Superskalar Ressource-inddeling:

```
ressourcer: Fe:2, De:2, Ex:2, Ag:1, Me:1, Wb:2
```

```
load:  "Fe De Ag Me Wb"
store: "Fe De Ag Me"
andre: "Fe De Ex Wb" // betingede hop behøver ikke Wb
branch: "Fe De Ex" // kun med betinget hop
```

Forkortelsen "Ag" står for "Address generate" som så erstatter brugen af den generelle ALU til at beregne adresser ved lagertilgang.

Der kan nu godt være 2 Fe på samme tid

```
aritmetisk op:
depend(Ex,Rs1),depend(Ex,rs2),depend(Ex,rd),produce(Wb,rd)
```

Dette vil sikre at der maximalt er en instruktion for hvert register i trinnene fra Ex og frem ad gangen. Det udelukker forkert skrive rækkefølge og det sikrer at operand-referencer er unikke.

```
                              0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
0:      lw   x12,0(x11)       Fe De Ag Me Wb
4:      addi x11,x11,4        Fe De Ex Wb
8:      sw   x12,0(x10)          Fe De Ag Me
C:      addi x10,x10,4           Fe De Ex Wb
10:     bne  x11,x15,0              Fe De Ex
0:      lw   x12,0(x11)                    Fe De Ag Me Wb
4:      addi x11,x11,4                     Fe De Ex Wb
8:      sw   x12,0(x10)                       Fe De Ag Me
C:      addi x10,x10,4                        Fe De Ex Wb
10:     bne  x11,x15,0                           Fe De Ex
0:      ...                                            Fe De Ex....
```

Her ser man at 2 instruktioner `lw` og `addi` begge kan være i fetch fasen på samme tid.