CS-A1120 Ohjelmointi 2

Aalto-yliopisto
Perustieteiden
korkeakoulu

Luento 4
kevät 2026

Sanna Suoranta (suomeksi, AS2) ja
Lukas Ahrenberg (englanniksi, T1)

16.3.2026

https://presemo.aalto.fi/o2fi2026

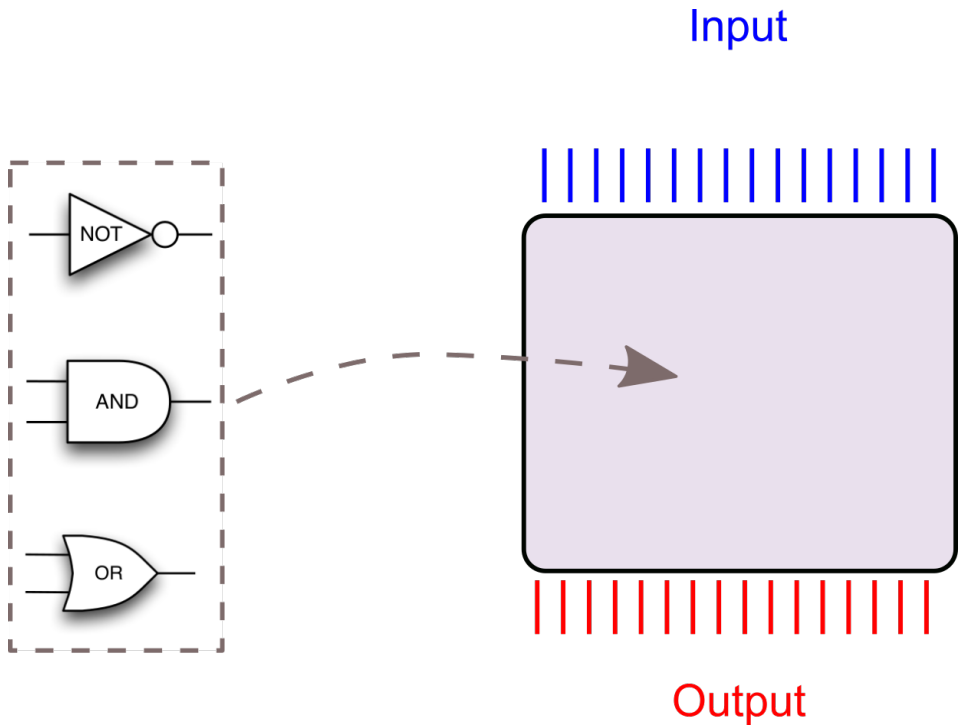
Tänään: Sekvenssilogiikka

https://presemo.aalto.fi/o2fi2026

Aalto-yliopisto
Perustieteiden
korkeakoulu

Tähän mennessä – voimme rakentaa koneen, joka laskee biteillä

- **Käytimme kombinaatiologiikan rakennuspalikoita**



- **Riittääkö tämä ohjelmoitavan koneen rakentamiseksi?**

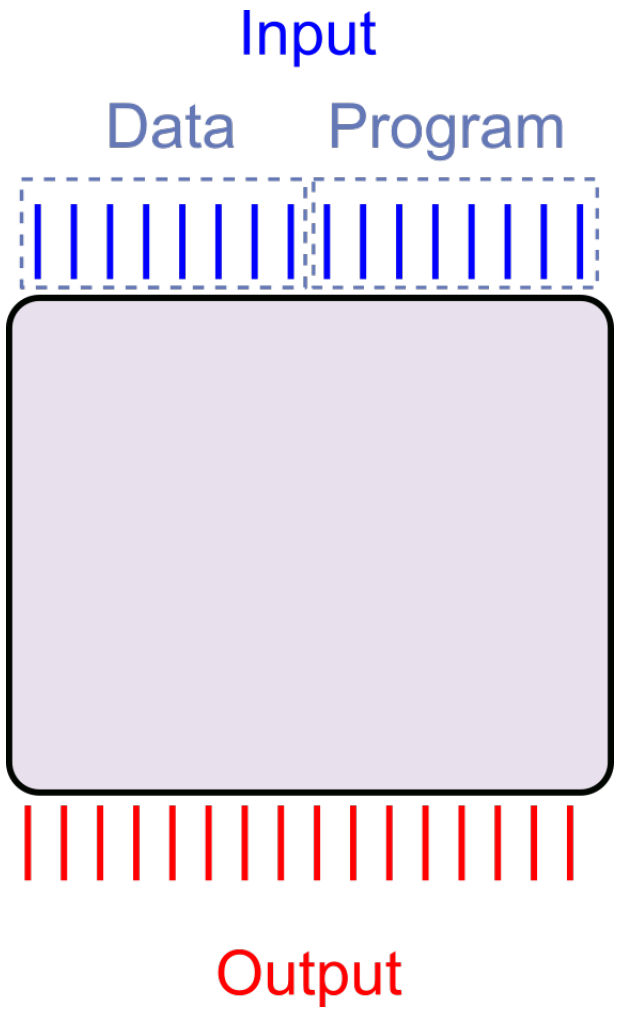
Mitä ohjelmoitavuus (programmable) tarkoittaa?

https://presemo.aalto.fi/o2fi2026

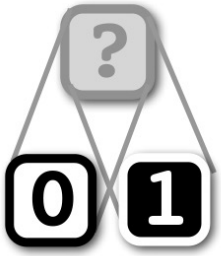
Kierroksen 4 oppimistavoitteet

- **Tämän viikon jälkeen,**
  - Tunnet kombinaatiologiikan rajoitukset
  - Osaat kertoa esimerkkejä kuinka takaisinkytkennän (feedback) lisääminen kombinaatiologiikkaan luo sekvenssilogiikan
  - Sinulla on ajatuksia ohjelmoitavuuden konseptista
  - Osaat selittää tilan (state) perusperiaatteet
  - Osaat rakentaa (käyttäen minilog-pakettia) yksinkertaisen laskentalaitteen, jonka tila kehittyy
  - Sinulla on käsitys kuinka tietokoneen tietovuo (data path) on rakennettu
  - Osaat kertoa kuinka tietokoneen muisti toimii

Mitä ohjelmoitavuus (programmable) tarkoittaa?



Onko kombinaatiologiikka riittävä ohjelmoitavan tietokoneen rakentamiseksi?



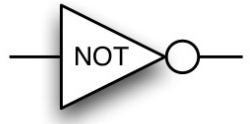
Syöte-elementit

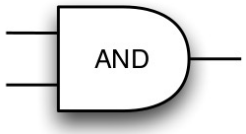
(input elements)

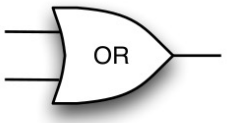
johdin

(wire)

Loogiset portit



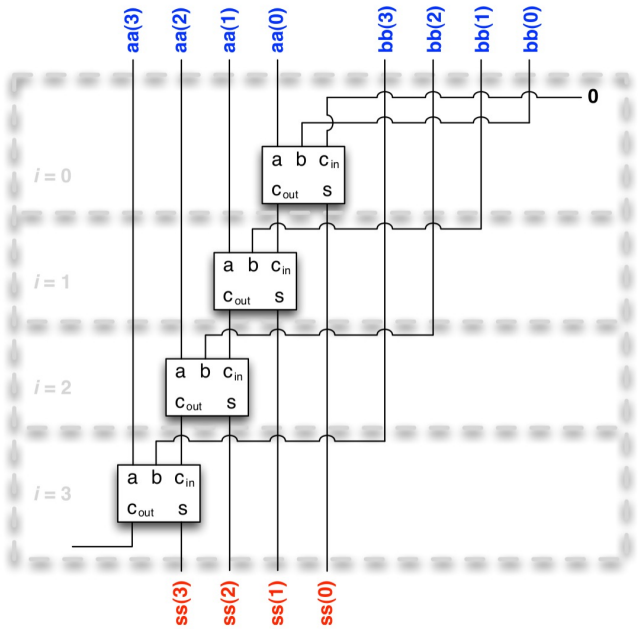




(logic gates)

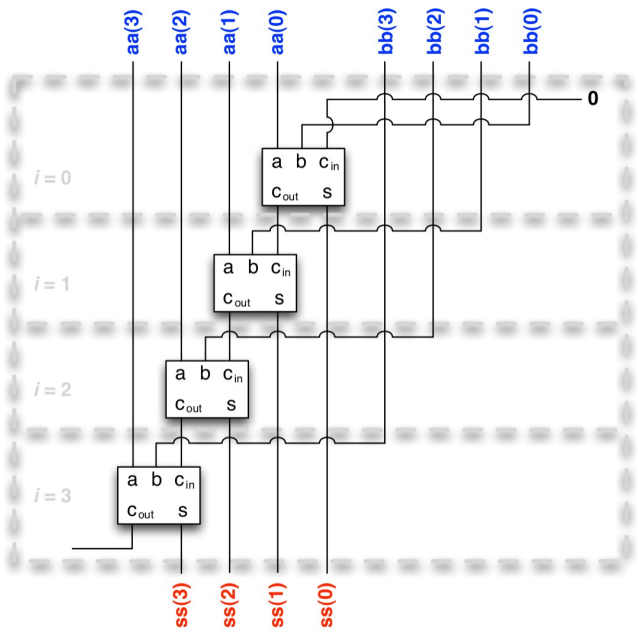
Ei ole (perustelut intuitioon pohjautuen “käsienheiluttelulla”)

- **Esimerkki: kahden `n`-bittisen luvun yhteenlasku**

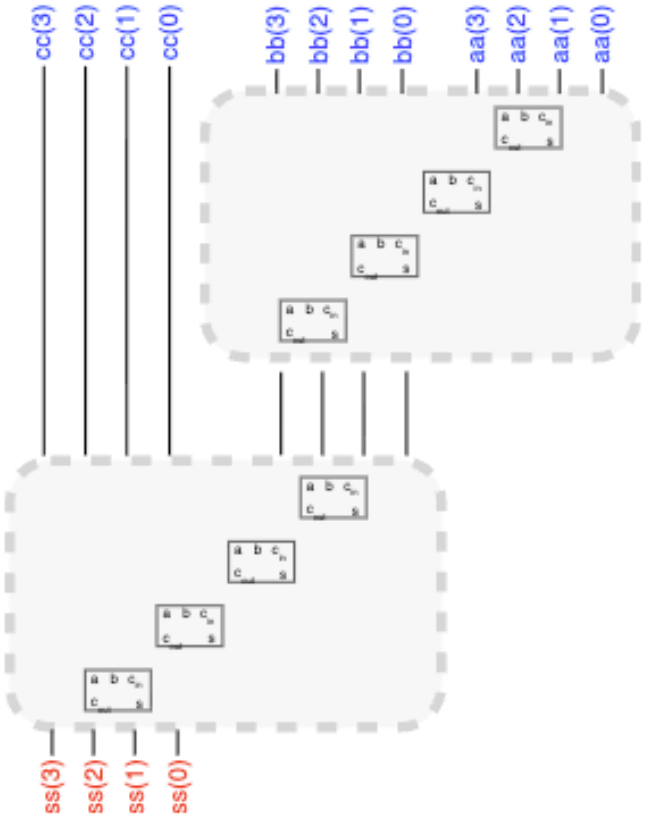


Ei ole (perustelut intuitioon pohjautuen “käsienheiluttelulla”)

- **Esimerkki: kahden `n`-bittisen luvun yhteenlasku**

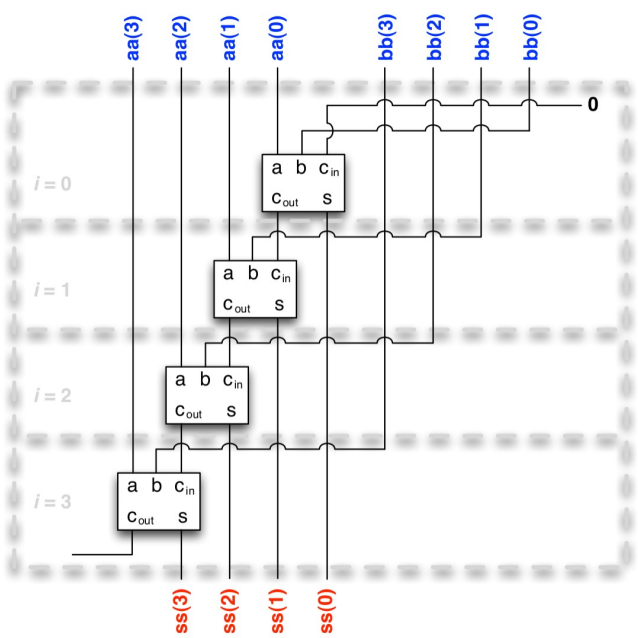


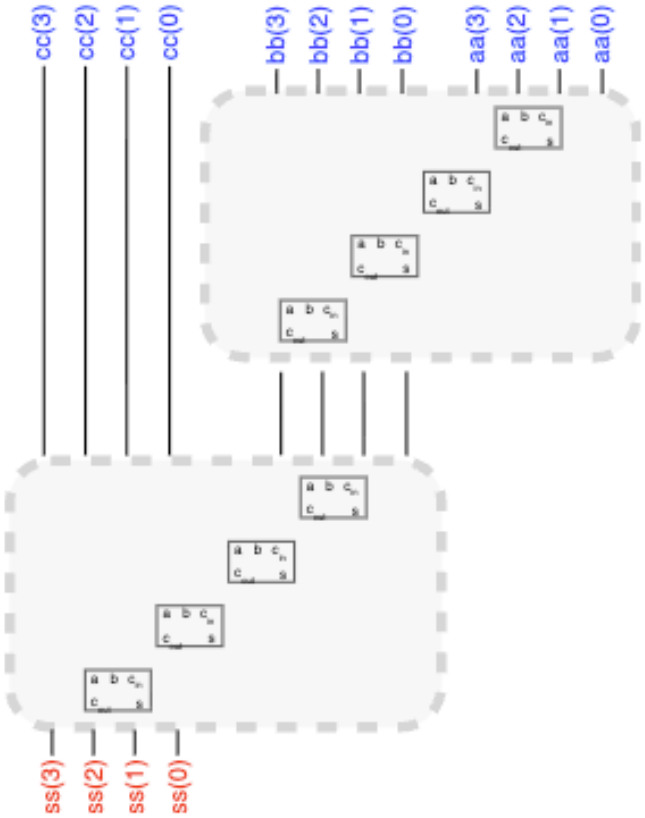
- **Kolmen `n`-bittisen luvun yhteenlasku**



Ei ole (perustelut intuitioon pohjautuen “käsienheiluttelulla”)

- **Entäpä m kappaletta n:n mittaisia lukuja…?**





Ei ole (perustelut intuitioon pohjautuen “käsienheiluttelulla”)

- **Esimerkki: kahden n-bittisen luvun yhteenlasku**
  - Tämä voidaan tehdä kombinaatiologiikalla (esim. viime viikolla)
- **Entäpä kolmen n-bittisen luvun yhteenlasku?**
  - Toki, kolme syötettä piirille: $x_1 + x_2 + x_3$
  - Tai kaksi summainta peräkkäin: $(x_1 + x_2) + x_3$
- **Mutta, piiri osaa tehdä vain sille annetun asian: lisätä toisiinsa kolme lukua**
  - jos kolmas luku on 0, lisäämme kaksi lukua
- **Entäpä yleinen lauseke käyttäen jotain toista operaattoria?**
  - Piiristä tulisi todella kompleksinen
- **Piiri ei voi tehdä mitään, mitä emme ole etukäteen ‘ajatelleet’**

Kombinaatiologiikan summain/vähennin (adder/subtracter)

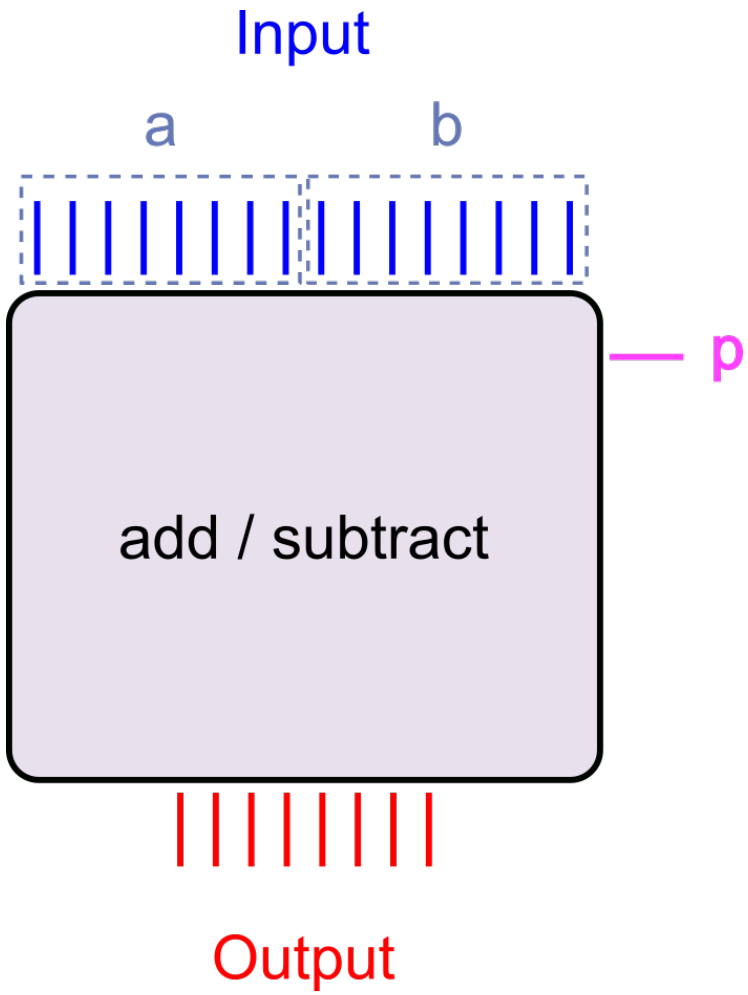
- **Kaksi syötesanaa ja yksi ’ohjelma-bitti’ operaation valitsemiseksi**

$$a \pm b$$

$$a + b, \text{if } p = 0$$

$$a - b, \text{if } p = 1$$

- **Voisimme rakentaa yhden summain-piirin, yhden vähennin-piirin sekä yhden kytkimen valinnan tekemiseksi (viime viikolla)**



Kombinaatiologiikan summain/vähennin (adder/subtracter)

- **Kolme syötesanaa ja kolme ’ohjelma-bittiä’ operaatioiden valitsemiseksi**

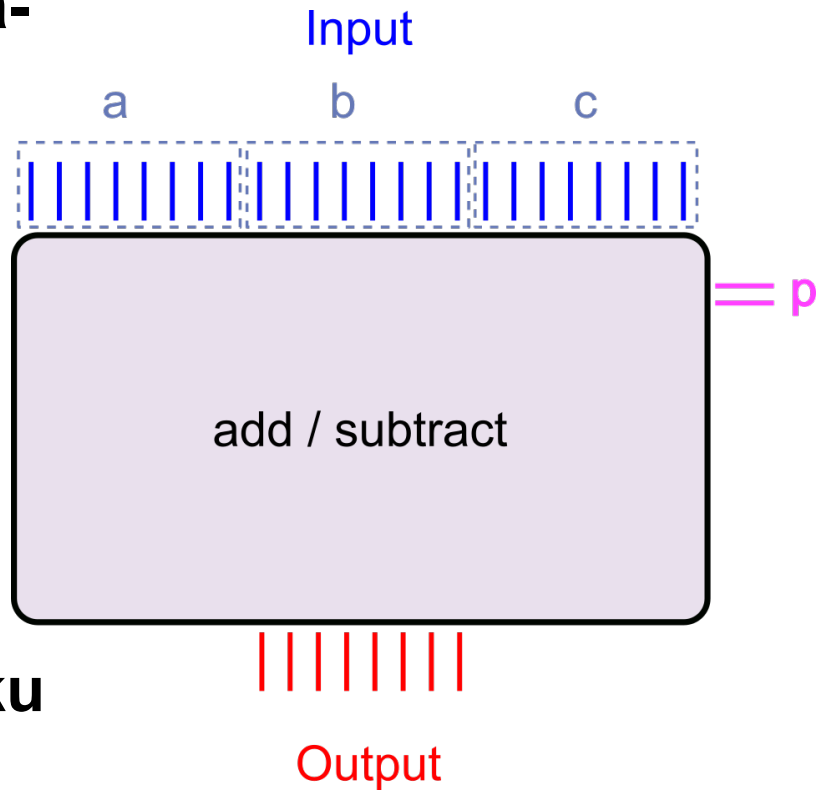
$$(a \pm b) \pm c$$

$$(a + b) + c, \text{if } p = 00$$

$$(a + b) - c, \text{if } p = 01$$

$$(a - b) + c, \text{if } p = 10$$

$$(a - b) - c, \text{if } p = 11$$



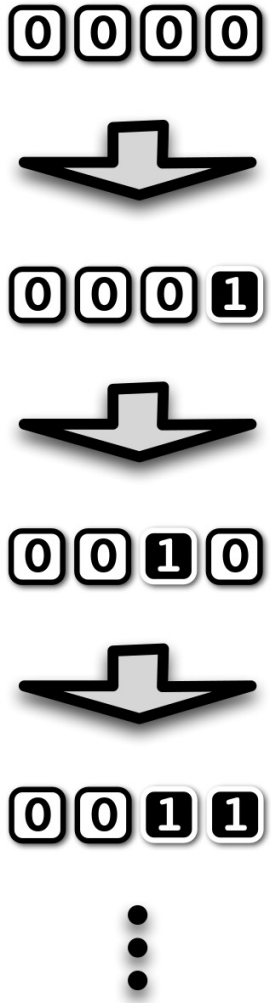
- **Onnistuisi vielä, mutta jokainen polku pitäisi rakentaa erikseen!**
  - Syötteiden määrä kasvaa

Mikä on “yksinkertaisin” kone, jota EI voi rakentaa kombinaatiologiikalla?

https://presemo.aalto.fi/o2fi2026

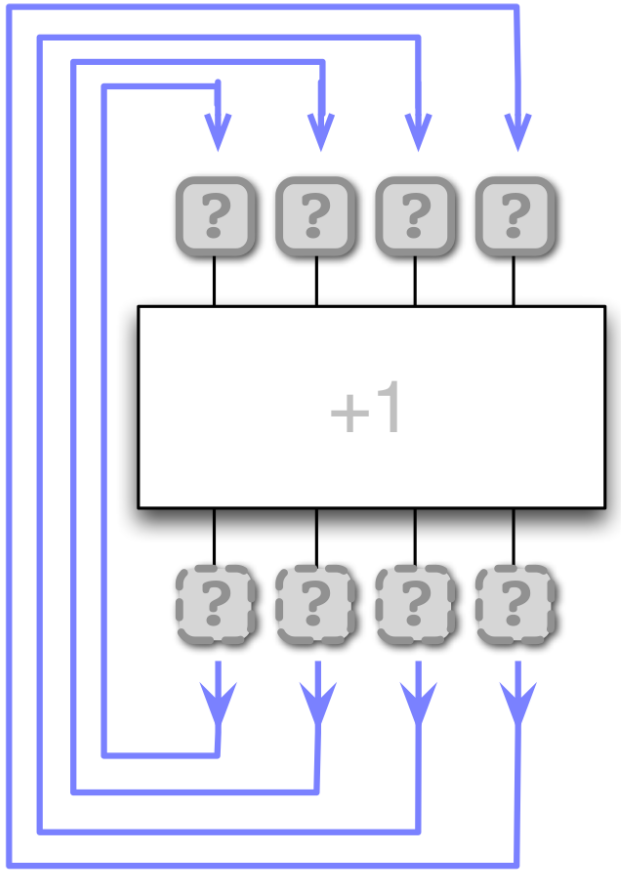
Mikä on “yksinkertaisin” kone, jota EI voi rakentaa kombinaatiologiikalla?

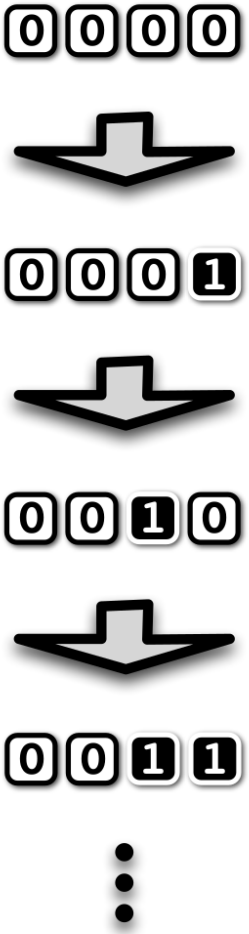
- **Esim. kone, joka kasvattaa laskuria (counter) yhdellä joka kerta, kun ‘nappia painetaan’**
- **Koneen pitäisi ’muistaa’ edellinen luku**
  - Sillä on nykyinen tila (state)
- **Mutta mikä on ‘nappi’?**
  - Mikä on periaate (principle), jolla se toimii (operate)?



Takaisinkytkentä (feedback)

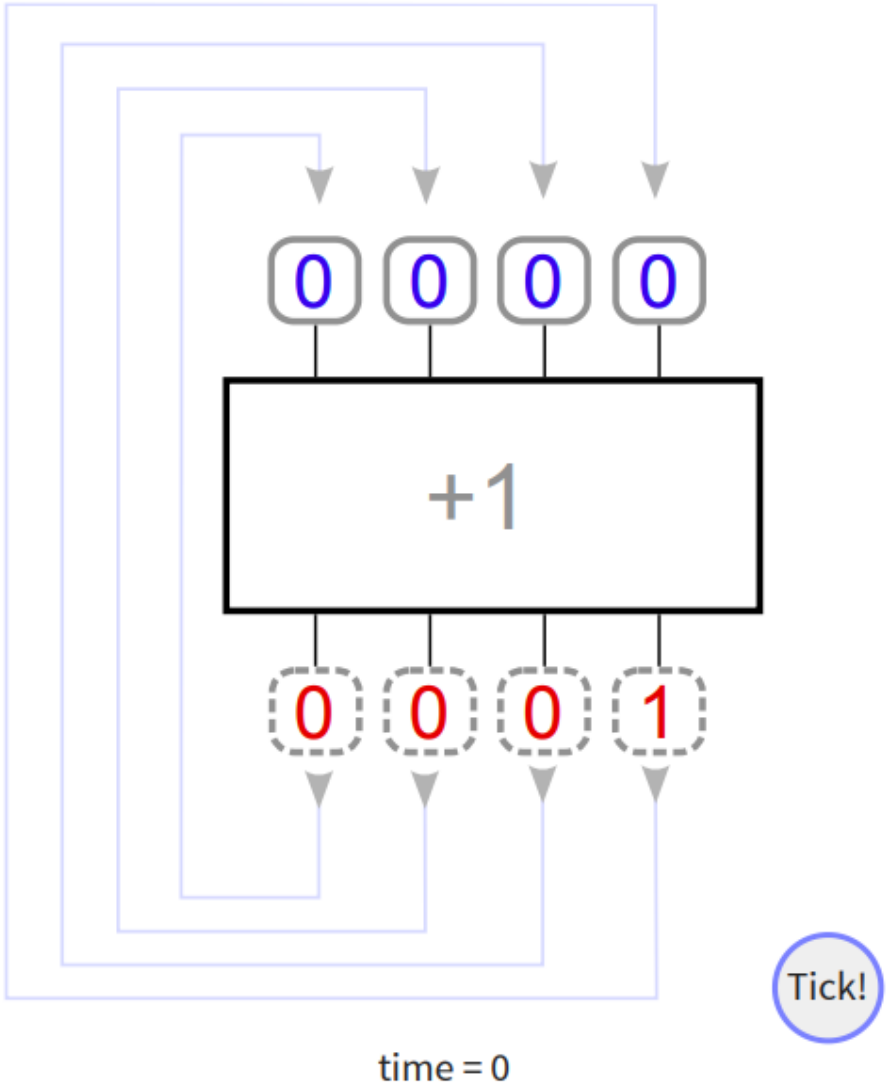
- **Ulostulo kytketään takaisin syötteisiin joka kerta, kun ‘nappia’ painetaan**
- **(Feedback = takaisinkytkentä, palaute)**





Kello (clock)

- **‘Napin’ painamisen voi automatisoida lisäämällä piiriin kellon**
- **Aina kun kello siirtyy askeleen eteenpäin, takaisinkytkentä aktivoituu**
  - Ulostulon arvot syötetään takaisin syötteiksi, mikä aiheuttaa piirin uudelleenarvioinnin (re-evaluate)



Kello (clock)

- **Kello ohjaa piirin tilaa päivittymään**
  - Mitä nopeampi kello, sen useampia päivityksiä (laskuja) per sekuntti
- **Modernien tietokoneiden kellotaajuus (clock frequency) on GHz taajuusalueella**
- **Mutta takaisinkytkentä vaatii energiaa (ja tuottaa lämpöä)**
- **Suurin osa tietokonearkkitehtuureista perustuu synktroniseen takaisinkytkentään**
  - On mahdollista tehdä epäsynkroninen takaisinkytkentä (ilman kelloa), mutta tämä on aika harvinaista

- **Värähtelypiiri (electronic oscillator) toteuttaa kellon laitetasolla**

https://en.wikipedia.org/wiki/Electronic_oscillator

Kuva: By User Smial on de.wikipedia - Own work, CC BY-SA 2.0 de, Link

Sekvenssilogiikan rakennuspalikat

syöte-elementit

(input elements)

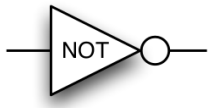
johdin

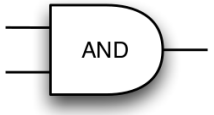
(wire)

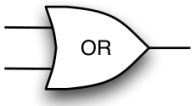
(clock triggered feedbacks)

kellon laukaisema takaisinkytkentä

loogiset portit



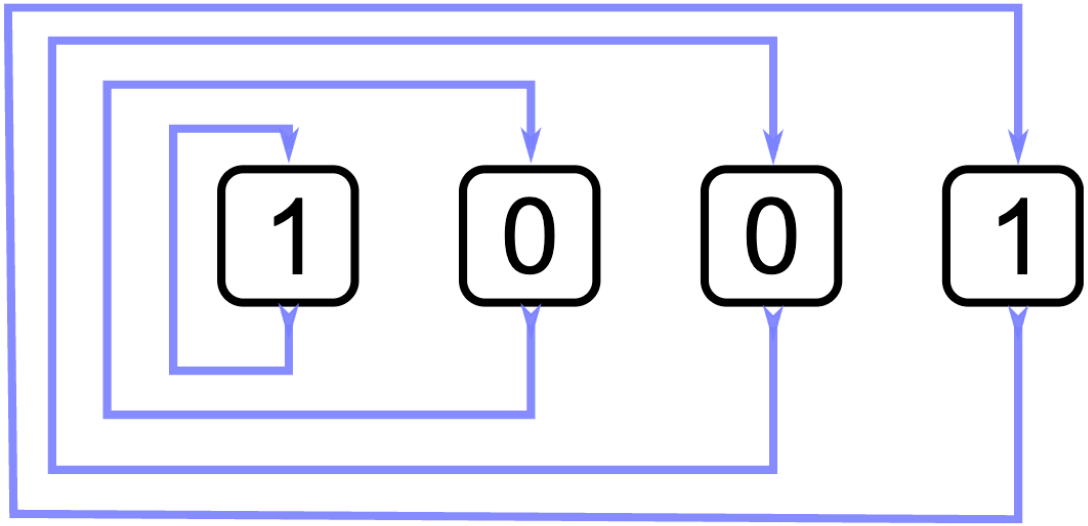




(logic gates)

Seuraava tila – tapaus 1?

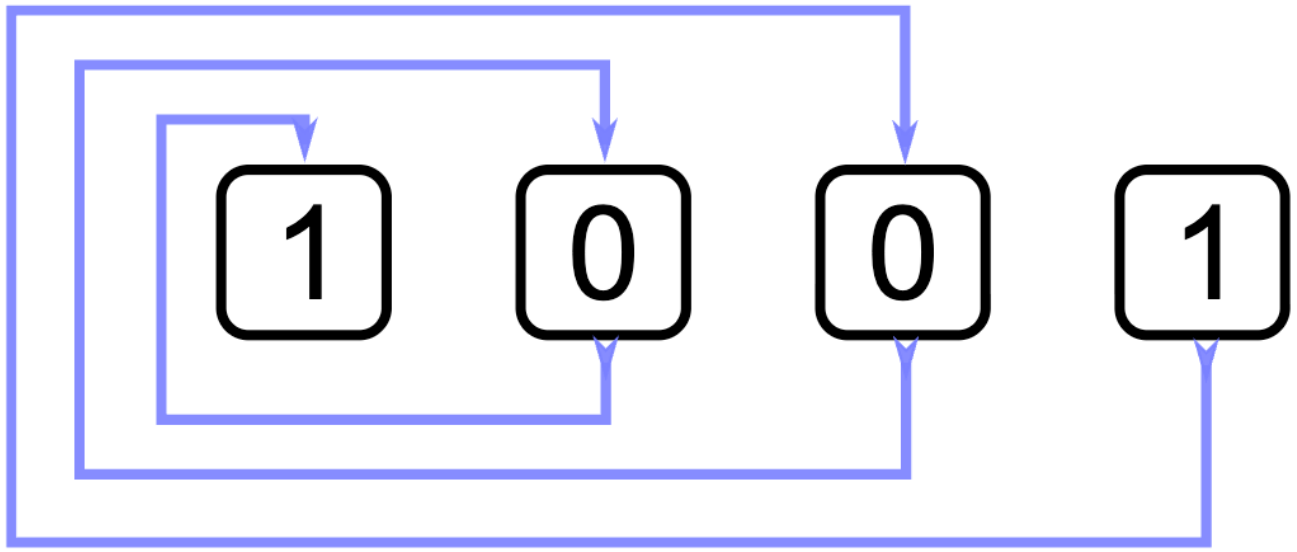
- **Mikä on tämän neliporttisen väylän uusi arvo sen jälkeen, kun takaisinkytkentä on lauennut kerran? (vasemmalta oikealle)**
  - Violetit nuolet merkitsevät takaisinkytkentää



https://presemo.aalto.fi/o2fi2026

Seuraava tila – tapaus 2?

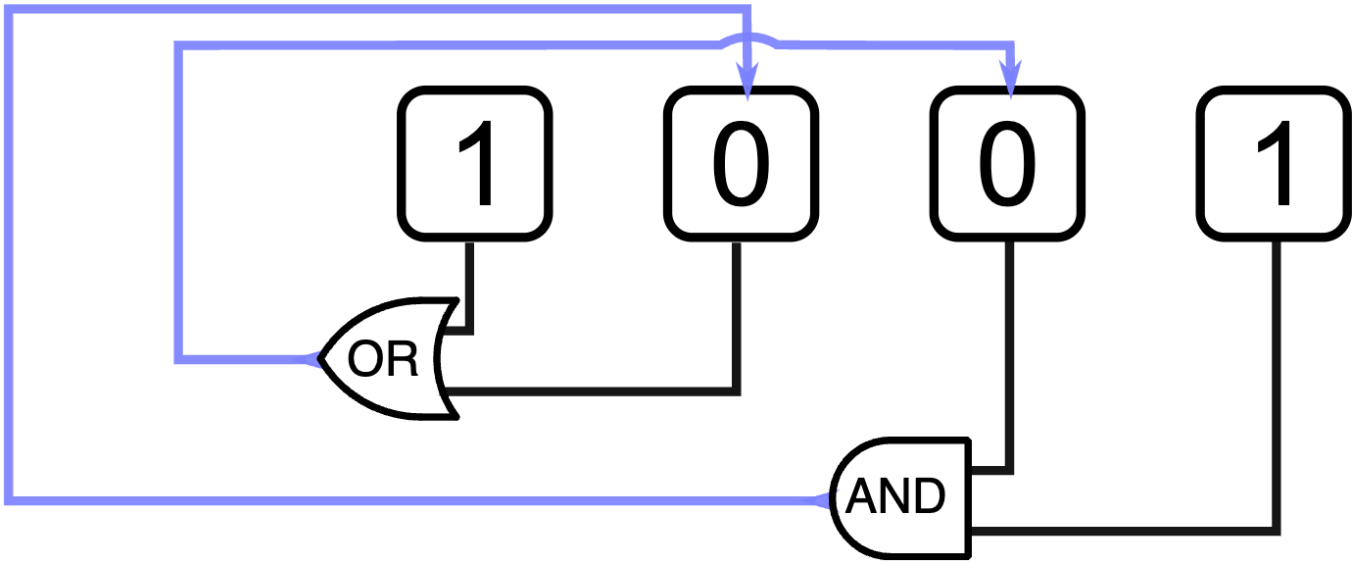
- **Entäs tämä yhden takaisinkytkentäkierroksen eli kellotuksen jälkeen? (vasemmalta oikealle)?**



https://presemo.aalto.fi/o2fi2026

Seuraava tila – tapaus 3?

- **Mikä on tämän seuraava tila?**



https://presemo.aalto.fi/o2fi2026

Onko tämä riittävää ohjelmoitavan tietokoneen rakentamiseksi?

Ihme kyllä – ON!

Muutama käytännön asia pitää vielä suunnitella, mutta periaatteessa sekvenssilogiikka riittää!

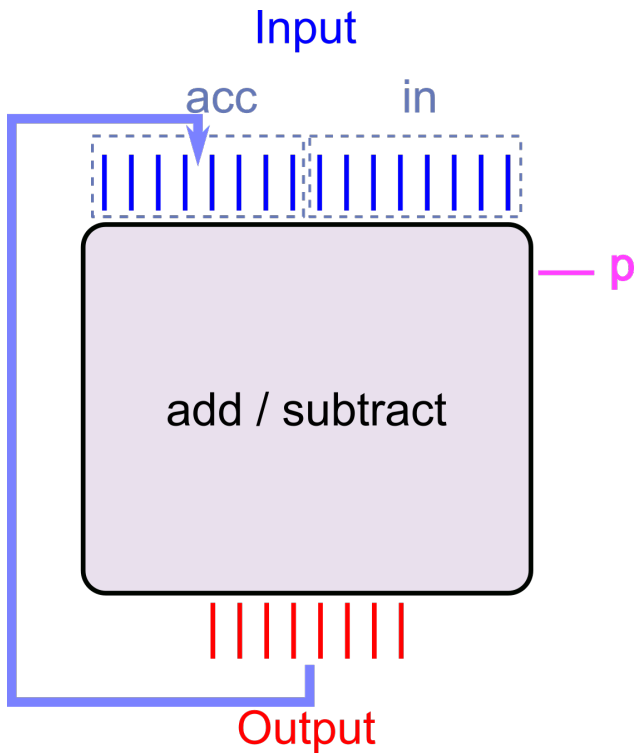
Aalto-yliopisto
Perustieteiden korkeakoulu

Miksi?

- **Takaisinkytkentä mahdollistaa sen, että piiri ylläpitää tilaa (state) eli se ‘muistaa’**
- **Logiikka mahdollistaa sen, että piirin tila voi kehittyä (evolve)**
  - Äärelliset tilakoneet (finite state machines) voivat kuvata prosessin mahdolliset toimintapolut (tästä lisää Laskennan teoria –kurssilla)
- **Toiminnallisuus määritellään (configure) eikä vain vaihdella (switch)**

Sekvenssilogiikan summain-vähennin (käsienheilutusversio)

- **Akkumulaattori (accumulator) voi laskea minkä tahansa sekvenssin** $x_1 \pm x_2 \pm x_3 \pm \ldots$:
  1. Aseta acc olemaan x_1, in olemaan x_2 ja p olemaan operaattori (0 yhteenlaskulle ja 1 vähennyslaskulle)
  2. Kellota piiri → acc on nyt edellisten kahden luvun yhteen/vähennyslaskun tulos
  3. Aseta in olemaan x_3 ja p olemaan operaattori
  4. Kellota piiri → acc:iin päivitetään uusi tulos
  5. Jne.



- **Askeleet 1, 3, 5, … ovat ‘ohjelma’**

`minilog` – sekvenssilogiikkaa Scalalla

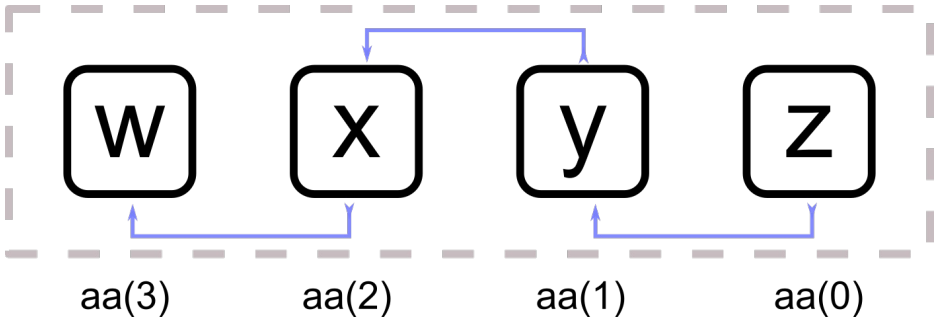
- **Sekvenssilogiikkaa varten meidän pitää päivittää `tinylog`-pakettia**
  - `Gate`- ja `Bus`-olioiden pitää olla osa `Circuit`-oliota, ikäänkuin piirikortteja (jotta ne ovat tietoisia samasta kellon laukaisemasta takaisinkytkennästä)
  - Tätä oliota kutsutaan `Gate`:n tai `Bus`:in isännäksi `host`
- **Takaisinkytkentä voidaan rakentaa mistä vaan portista, mutta se voidaan kytkeä vain `inputElement`-olioon**
- **Tämän viikon harjoitusten Scala-paketista löytyy nämä toteuttava `minilog`-paketti**
  - Lisätietoja paketista kurssimateriaalissa: https://a1120.cs.aalto.fi/2024/notes/round-sequential-logic--seq.html

`minilog` – sekvenssilogiikkaa Scalalla

- **`minilog`-paketista löytyy XOR-operaattori**
  - XOR kahden portin välillä: toimii sekä `^^` että `+`
- **Rajapinta:**
  - `val c = new Circuit()` luo uuden piirin
  - `val g = c.input()` (`inputElement`) luo uuden syöte-elementin piirille `c`
  - `val bb = c.input(8)` luo 8-bittisen syöte-elementin piirille `c`
  - `g.host` kertoo mihin piiriin `g` kuuluu
  - `a.buildFeedbackFrom(b)` tekee takaisinkytkennän `b`:stä `a`:n syöte-elementtiin(/elementteihin)
  - `c.clock()` laukaisee piirin `c` kellon

`minilog`-esim.: 4 porttia, jossa takaisinkytkentä portista `n` porttiin `n+1`

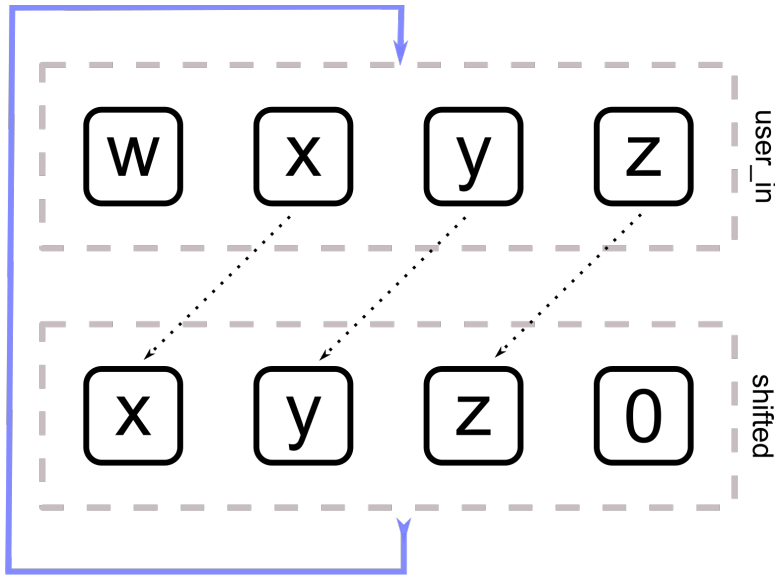
```
import minilog._
// Always need a circuit
val c = new Circuit()
// Create the bus of input elements
val aa = c.inputs(4)
// Feedback TO gate at index 1
// FROM gate at index 0
aa(1).buildFeedbackFrom(aa(0))
// From index 1 -> index 2
aa(2).buildFeedbackFrom(aa(1))
// You should use a loop for this
aa(3).buildFeedbackFrom(aa(2))

// Build a trigger to visualise
val t = new Trigger(c)
t.watch("aa", aa.reverse)
t.go()
```



- Mitä tapahtuu, kun asetat portin 0 olemaan ykkönen ja kellotat piirin?
- Entä mikä tahansa muu portti ykköseksi?
- Miksi?

minilog-esim.: takaisinkytkentä väylien välillä

```
def buildShiftUpOne(user_in : Bus) : Bus =
  // Shifting 'up' means that the gate at the
  // 'highest' index disappears (is dropped)
  // and a False gate is inserted at the lowest.
  // Because Bus is a sequence we can drop and prepend.
  val shifted = new
  Bus(user_in.host.False
    +: user_in.dropRight(1))
  // We also need to feed back to the input so that
  // the operation can be repeated
  user_in.buildFeedbackFrom(shifted)
  // Return the new bus
  shifted
end buildShiftUpOne
```




```
// Let's test it
val c = new Circuit()
val aa = c.inputs(8)
val bb = buildShiftUpOne(aa)

val t = new Trigger(c)
t.watch("input",aa.reverse)
t.watch("shifted",bb.reverse)
t.go()
```

- **Piiri siirtää (shift) sanan yhden askeleen ‘ylöspäin’ joka kellonlöymällä**

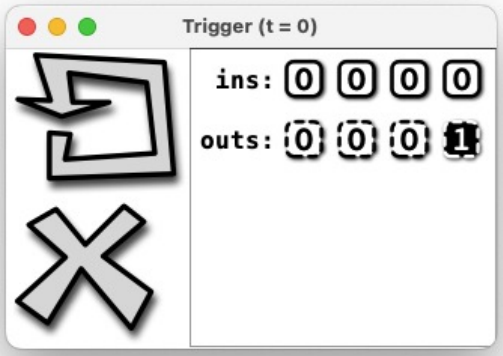
minilog-esim.: Askeltaja (incrementer)

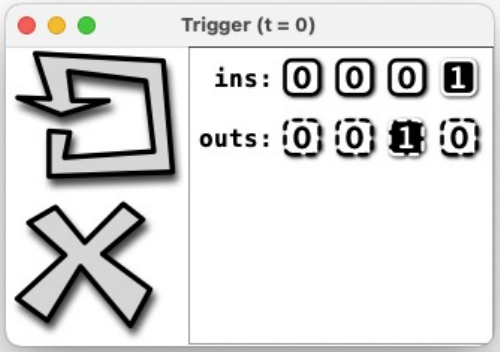
```
import minilog.* // Note: minilog not tinylog
def buildHalfAdder(ai: Gate, c: Gate) = (ai && c,
  (!ai && c)||(ai && !c))
def buildIncrementer(aa: Bus) =
  var c = aa.host.True // initial carry is true
  val r = new Array[Gate](aa.length)
  for i <- 0 until aa.length do
    val (c_out,s) = buildHalfAdder(aa(i), c)
    r(i) = s
    c = c_out
  end for
  new Bus(r.toSeq)
end buildIncrementer
```

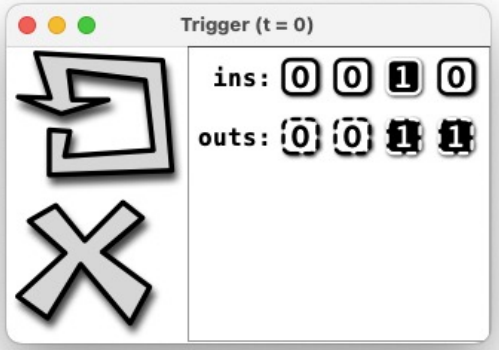
- **Takaisinkytkentää varten**

```
val n = 4
val c = new Circuit()
val ins = c.inputs(n)
val outs = buildIncrementer(ins)
ins.buildFeedbackFrom(outs)
```

`minilog`-esim.: Askeltaja (incrementer)





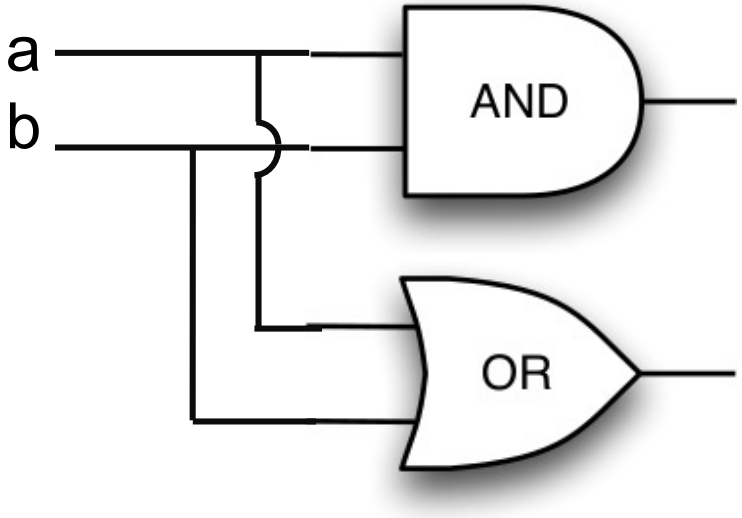


- **minilogTriggerin käynnistys:**

```
val t = new Trigger(c)
t.watch("ins", ins.reverse)
t.watch("outs", outs.reverse)
t.go()
```

- **Kombinaatiologiikan piiri ilman takaisinkytkentää: tämä kasvattaisi syötettä vain kerran**

(muistin virkistykseksi askeltajan ymmärtämiseksi: puolisummain, half-adder)



summa

carry
(muisti-
numero)

a	b	summa	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

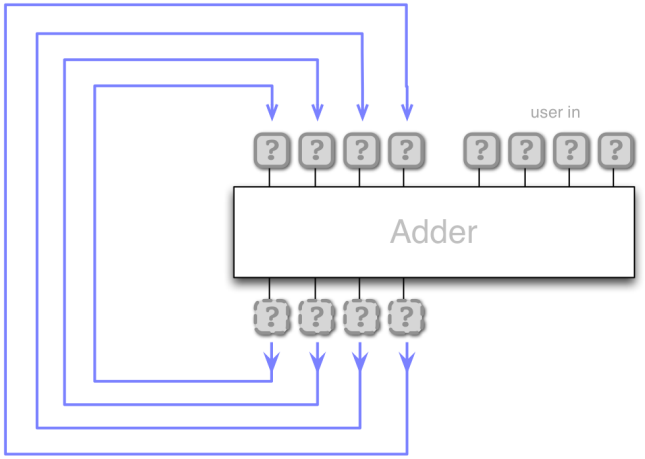
Minkälaisen piirin tämä funktio luo?

```
import minilog.*

def f(aa: Bus, bb: Bus) : Bus =
  val pairs = aa.zip(bb)
  var c : Gate = aa.host.False
  // Create sequence of gates using for-yield
  val ss = for (a,b) <- pairs yield
    val s = a ^^ b ^^ c          // ^^ means XOR
    c = ( (a && b) || (a && c) || (b && c) )
    s
  // Convert sequence to bus and return
  new Bus(ss)
end f
```

https://presemo.aalto.fi/o2fi2026

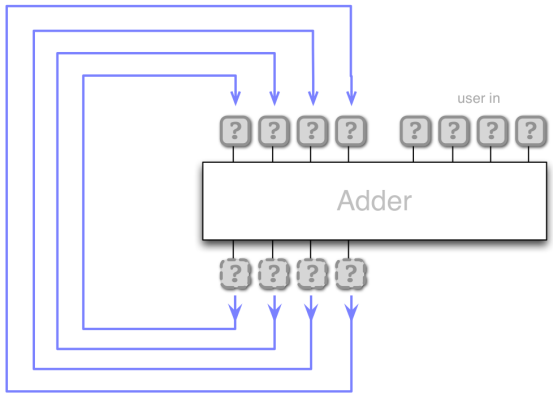
minilog-esim.: Akkumulaattori-summain (accumulator-adder)

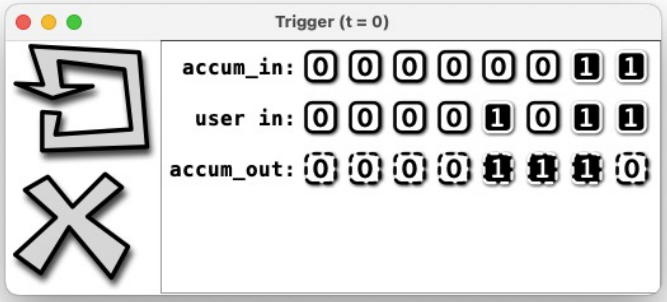
```
def buildAdder(aa: Bus, bb: Bus) : Bus =
  val pairs = aa.zip(bb)
  var c : Gate = aa.host.False
  // Create sequence of gates using loop
  val sum = for (a,b) <- pairs yield
    // Construct s gate for this bit
    val s = a ^^ b ^^ c          // ^^ means XOR
    // Update carry
    c = ( (a && b) || (a && c) || (b && c) )
    s // Return s gate
  new Bus(sum)
end buildAdder
```



`minilog`-esim.: Akkumulaattori-summain (accumulator-adder)

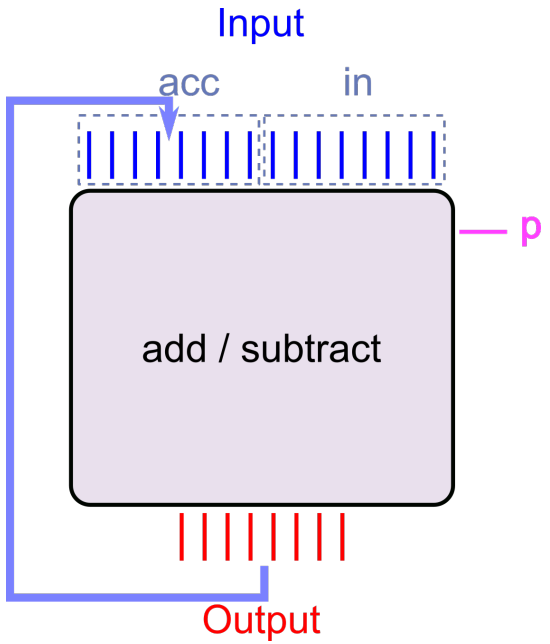
```
val n          = 8
val c          = Circuit()
val accum_in   = c.inputs(n)
val user_in    = c.inputs(n)
val accum_out = buildAdder(accum_in, user_in)

accum_in.buildFeedbackFrom(accum_out)

val t = Trigger(c)
t.watch("accum_in",  accum_in.reverse)
t.watch("user_in",   user_in.reverse)
t.watch("accum_out", accum_out.reverse)
t.go()
```





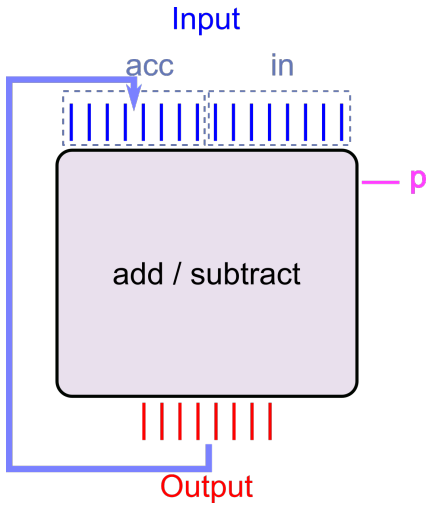
`minilog`-esim.: Akkumulaattori-summain-vähennin (accumulator-adder-subtracker)

```
def buildSubtracter(aa: Bus, bb: Bus) : Bus =
  val pairs = aa.zip(bb)
  var c : Gate = aa.host.True
  // Create sequence of gates using loop
  val diff = for (a,b) <- pairs yield
    // Construct s gate for this bit
    val s = !(a ^^ b ^^ c)          // ^^ means XOR
    // Update carry
    c = ( (a && !b) || (a && c) || (!b && c) )
    s // Return s gate
  // Return a new bus from sequence
  new Bus(diff)
end buildSubtracter
```



`minilog`-esim.: Akkumulaattori-summain-vähennin (accumulator-adder-subtracker)

```
val n          = 8
val c          = Circuit()
val accum_in   = c.inputs(n)
val user_in    = c.inputs(n)
val sel        = c.input()
// Adder same as on previous slide
val res_add    = buildAdder(accum_in, user_in)
val res_sub    = buildSubtracter(accum_in, user_in)
// Switch operator using the sel gate
val accum_out = (res_add && !sel) | (res_sub && sel)

accum_in.buildFeedbackFrom(accum_out)
```

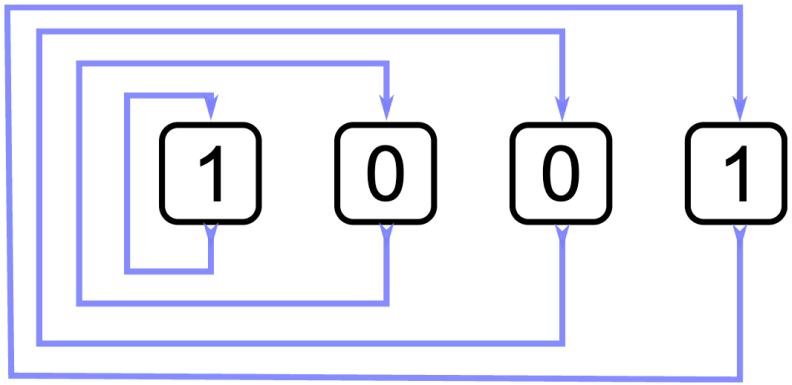


Ohjelmabitin kääntö vaikuttaa heti, mikä ei ehkä ole ihan intuitiivista.

```
val t = new Trigger(c)
t.watch("accum_in" , accum_in.reverse)
t.watch("sel" , sel)
t.watch("user_in" , user_in.reverse)
t.watch("accum_out" , accum_out.reverse)
t.go()
```

Kun pystymme pitämään yllä tilan, saadaan MUISTI

Muistimoduuli (yhdelle sanalle)

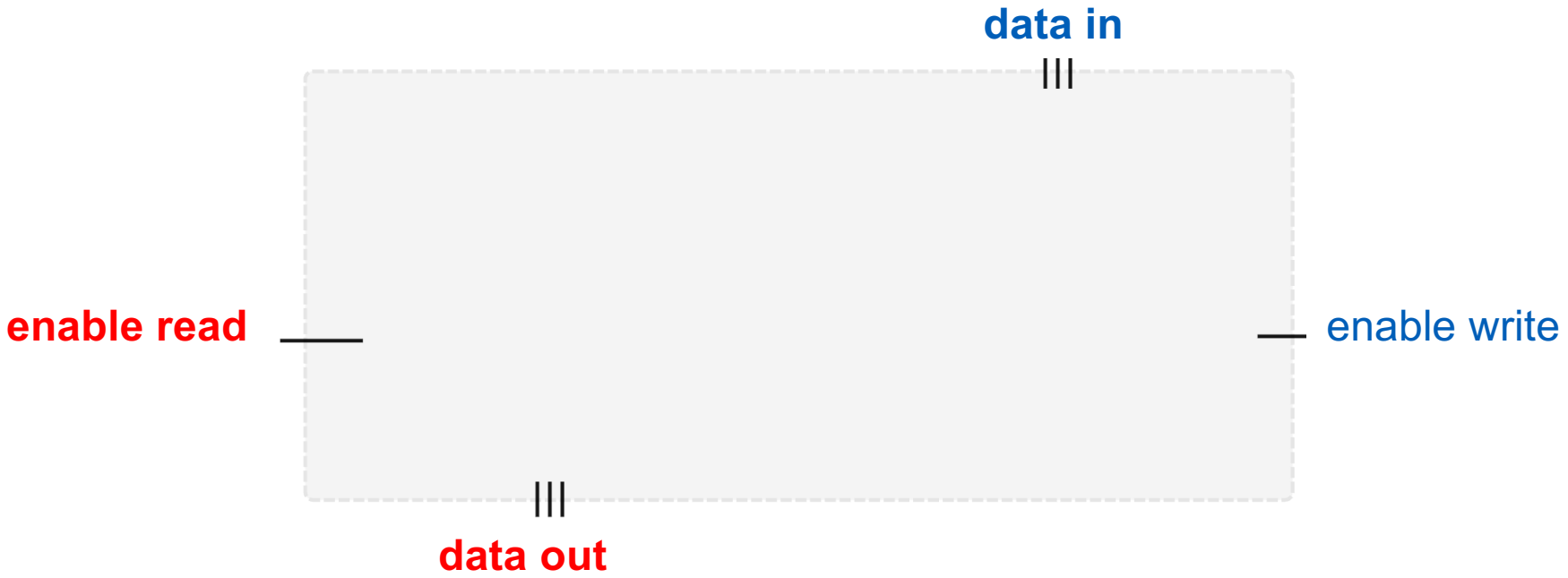


- **Tarve:**
  - Sanan (väylän) tilan pitäminen
  - Tallennetun sanan lukeminen (read)
  - Uuden sanan kirjoittaminen (write)
- **Takaisinkytkentä auttaa tilan pitämiseen**
- **Kuinka voimme tallentaa sanan?**
- **Kuinka voimme hakea talletetun sanan?**

- **Tarvitsemme jotain muuta kuin yksinkertaisen väylän, joka syöttää bitit takaisin itseensä**

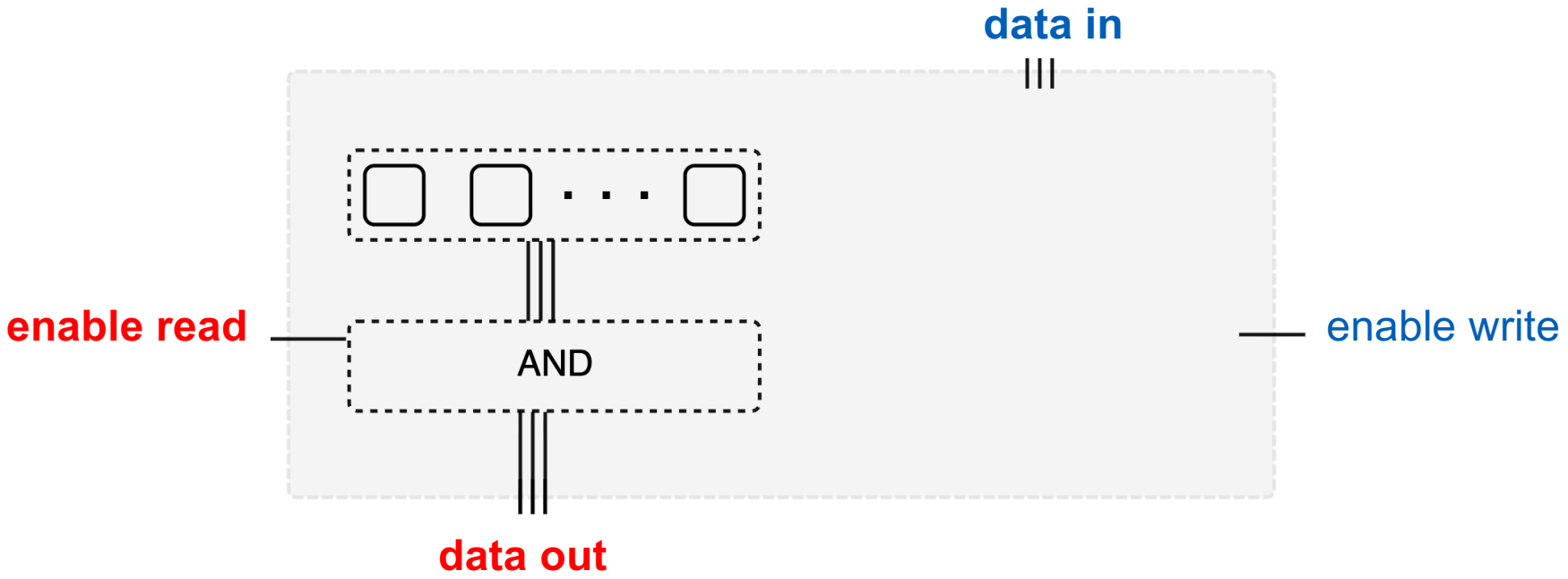
Muistimoduuli (yhdelle sanalle)

- Kun portti lue (read enable) on 1, data out –väylä välittömästi asetetaan muistiin tallennetuksi sanaksi
- Kun portti kirjoita (write enable) on 1 ja piiriä kellotetaan, sana väylästä data in kirjoitetaan muistiin



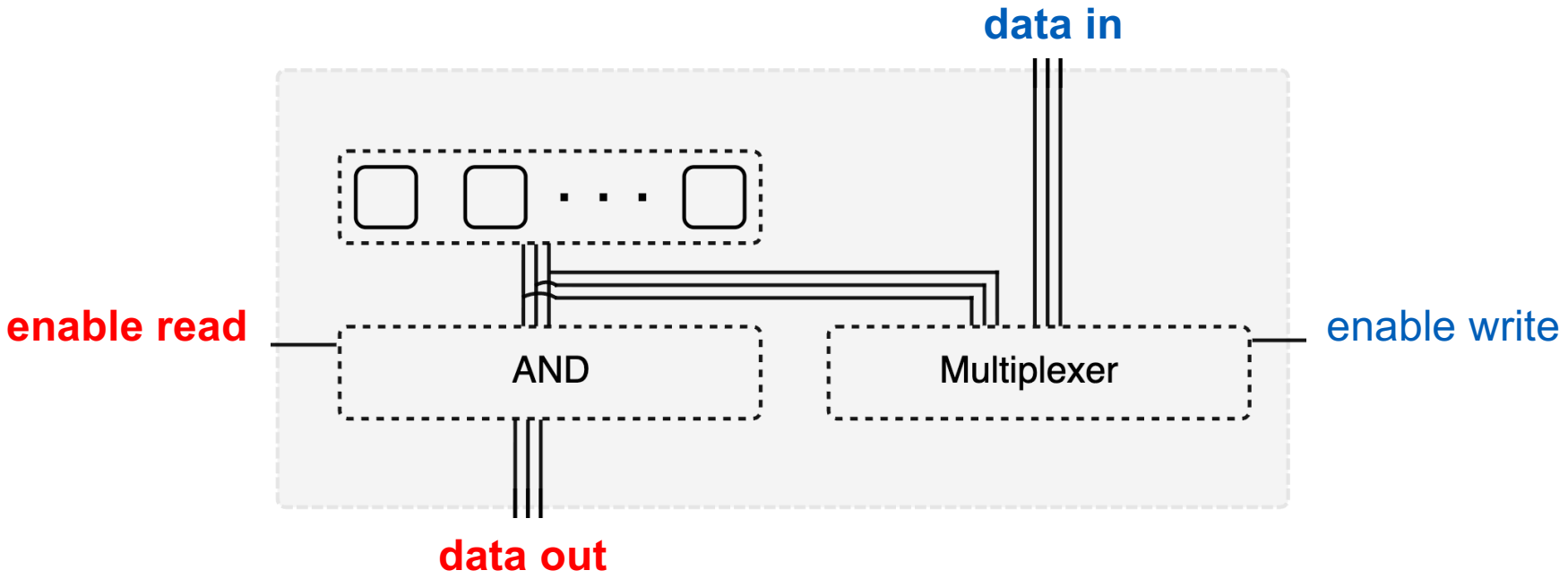
Muistimoduuli (yhdelle sanalle)

- Kun portti lue (read enable) on 1, data out –väylä välittömästi asetetaan muistiin tallennetuksi sanaksi
- Kun portti kirjoita (write enable) on 1 ja piiriä kellotetaan, sana väylästä data in kirjoitetaan muistiin



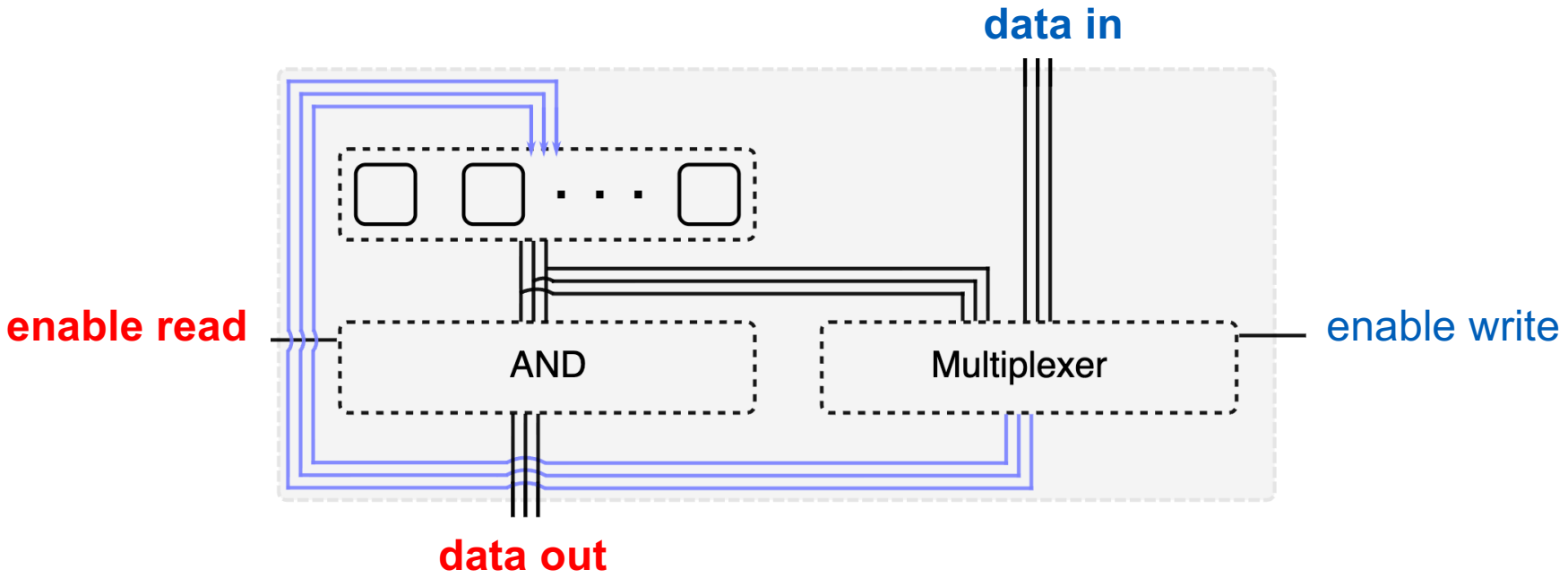
Muistimoduuli (yhdelle sanalle)

- Kun portti lue (read enable) on 1, data out –väylä välittömästi asetetaan muistiin tallennetuksi sanaksi
- Kun portti kirjoita (write enable) on 1 ja piiriä kellotetaan, sana väylästä data in kirjoitetaan muistiin



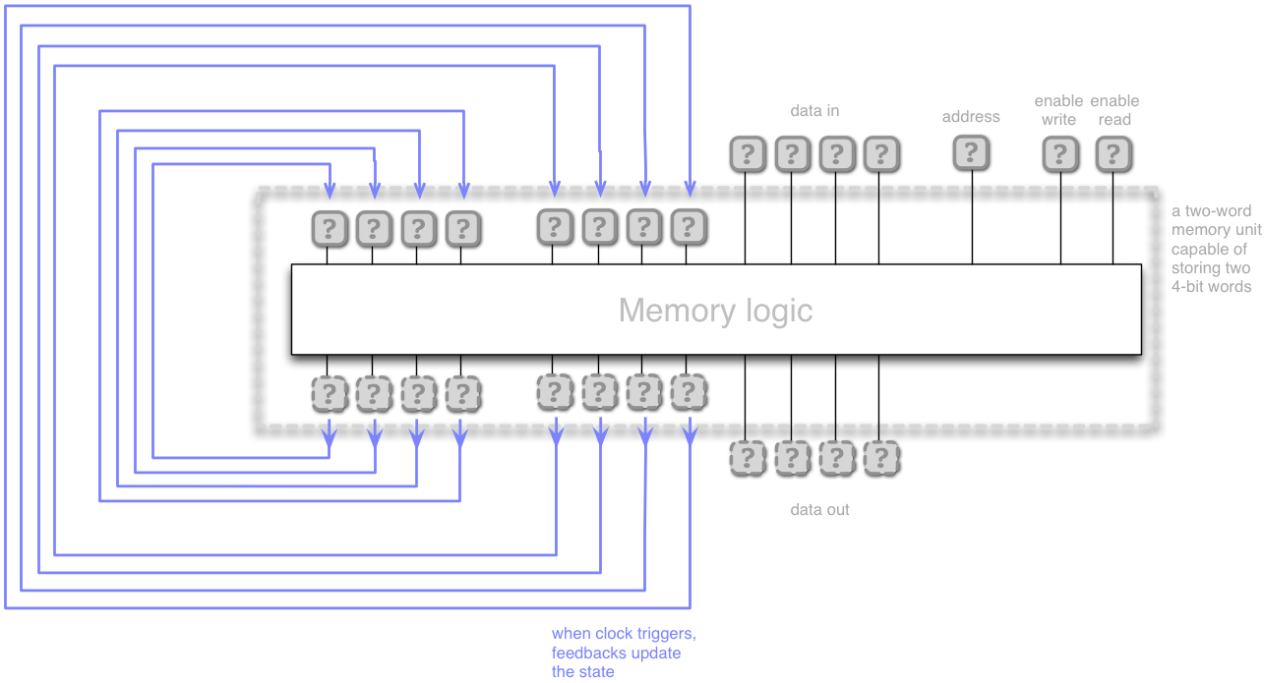
Muistimoduuli (yhdelle sanalle)

- Kun portti lue (read enable) on 1, data out –väylä välittömästi asetetaan muistiin tallennetuksi sanaksi
- Kun portti kirjoita (write enable) on 1 ja piiriä kellotetaan, sana väylästä data in kirjoitetaan muistiin



Kahden sanan muisti (konseptina)

- **Takaisinkytkentä mahdollistaa sen, että piiri “muistaa”**
- **Muistipankki (memory bank) on joukko väyliä, jotka syötetään takaisin itseensä**
- **Kun lisää sanan lukemisen ja kirjoittamisen, saadaan muisti (memory)**

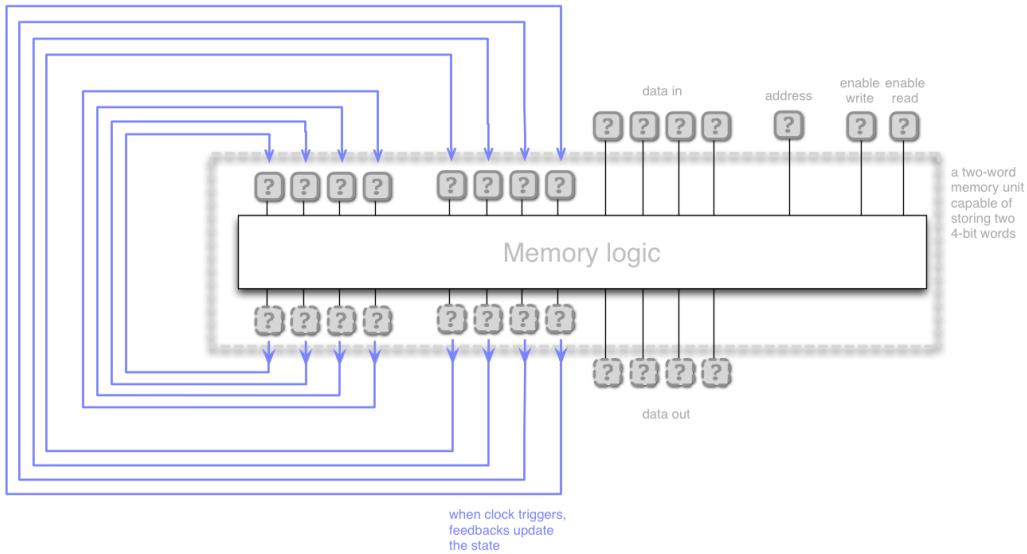


Address-väylä kertoo, kumpaa sanaa käsitellään
- Tässä yksi bitti, koska on vain kaksi sanaa

(tästä video)

Muisti

- **Vertaa Scalan `Buffer`iin tai `Array`'n**
- **`address` on muistin 'indeksi'**
- **`enable write` ja `enable read` pitäisikö meidän asettaa vai saada arvo**
  - Kun `enable write` on tosi, `data in` ylikirjoittaa `address`in osoittaman sanan
  - Kun `enable read` on tosi, `address`in osoittama sana kopioidaan `data out`:iin



Muisti

- **Muistimoduulilla on kaksi operaatiota:**
  - Muistin lukeminen (memory read)
  - Muistin kirjoitus (memory write)
- **Kaksi sanaa sisään:**
  - Muistin osoite, eli mihin muistissa olevaan sanaan operaatio vaikuttaa
  - Muistidata eli datasana, joka tallennetaan osoitteen osoittamaan paikkaan kirjoitus-operaatiossa
- **Yksi sana ulos:**
  - Luettu sana ulos luku-operaatiossa

Osoite-väylän leveys ja muistin koko

- **Osoite-väylän leveys rajoittaa osoiteavaruutta (address space), eli kuinka monta eri paikkaa muistissa on**

  https://presemo.aalto.fi/o2fi2026

- **Kuinka monta bittiä tarvitsemme osoitteeksi, kun muistipankkiin mahtuu 8 sanaa?**
  - (Vinkki: 1 bitti antaa 2 sanan muistin)
- **Jos meillä on 8 bitin osoiteavaruus ja sanan koko on 1 tavu, kuinka iso muisti meillä on?**
- **Monella koneella on nykyään sama osoiteväylä ja sanakoko, mutta tämä ei ole pakollista**
  - Esim Commodeore 64:ssa oli 16 bit osoite mutt 8 bit sanan pituus

ARMLET

Aloitetaan ohjelmoitavan prosessorin rakentaminen (viimeinkin)

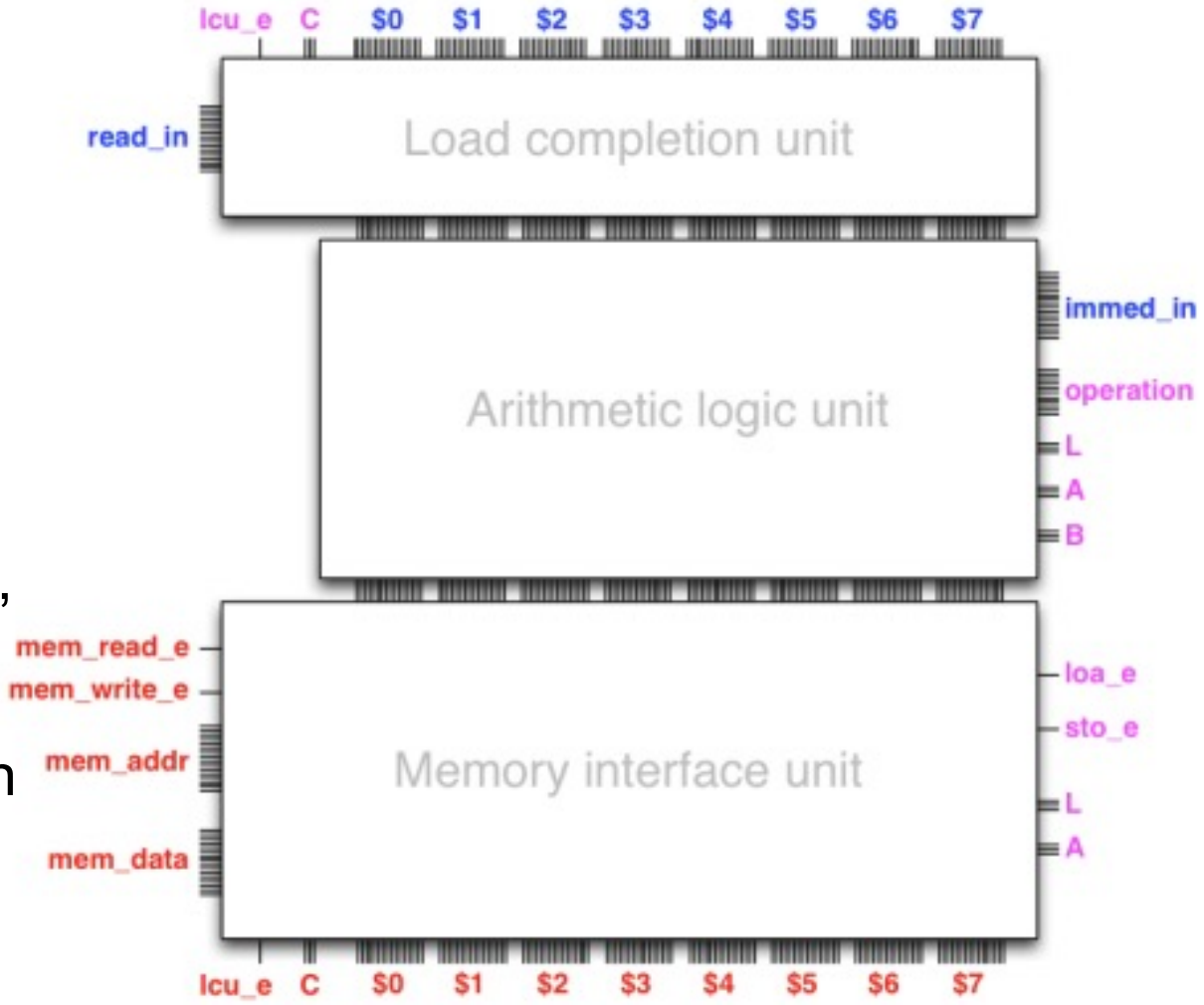
Armlet

- **128 KiB muisti, 16-bittinen prosessori, osoiteavaruus ja väylä**
  - yksinkertainen
- **Hyvin yksinkertainen rakenne (design)**
- **ARM-prosessorin insiprioma**
- **Noudattaa von Neumann (aka. Princeton) –arkkitehtuuria**
  - Muistissa sekä ohjelma että data, I/O-toiminta siirtelyyn
  - CPU (central processing unit) sisältää ALUn ja operoi datapolkua

ARM = Advanced RISC machines
RISC = reduced instruction set computer

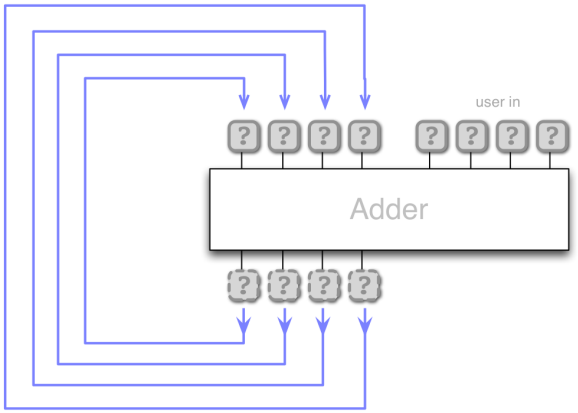
Datapolku (data path)

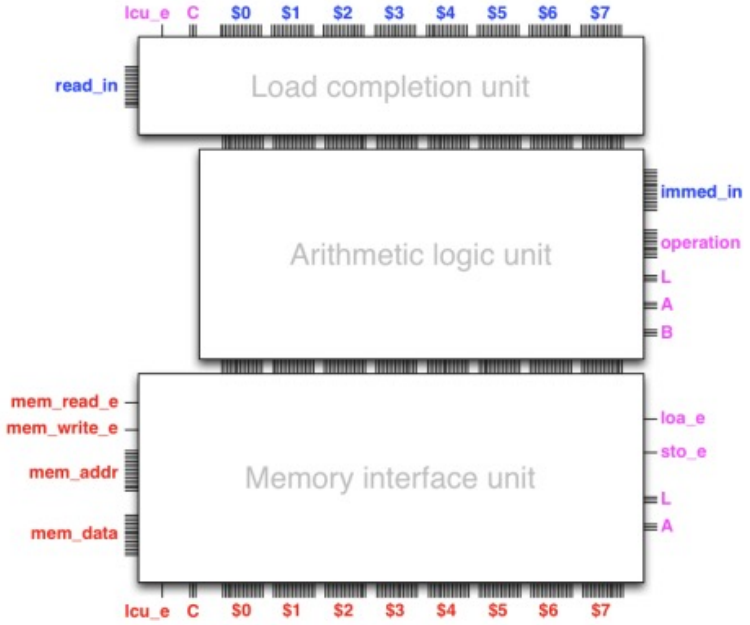
- **Mitä tarvitsemme?**
  - Sisäisen tilan (rekisterit, registers)
  - Kyvyn ladata tietoa prosessorille (latauksen viimeistely-yksikkö, load completion unit)
  - Kyvyn laskea (aritmeettislooginen yksikkö, arithmetic logic unit)
  - Kyvyn tallentaa tulos tietokoneen muistiin (muistin rajapintayksikkö, memory interface unit)



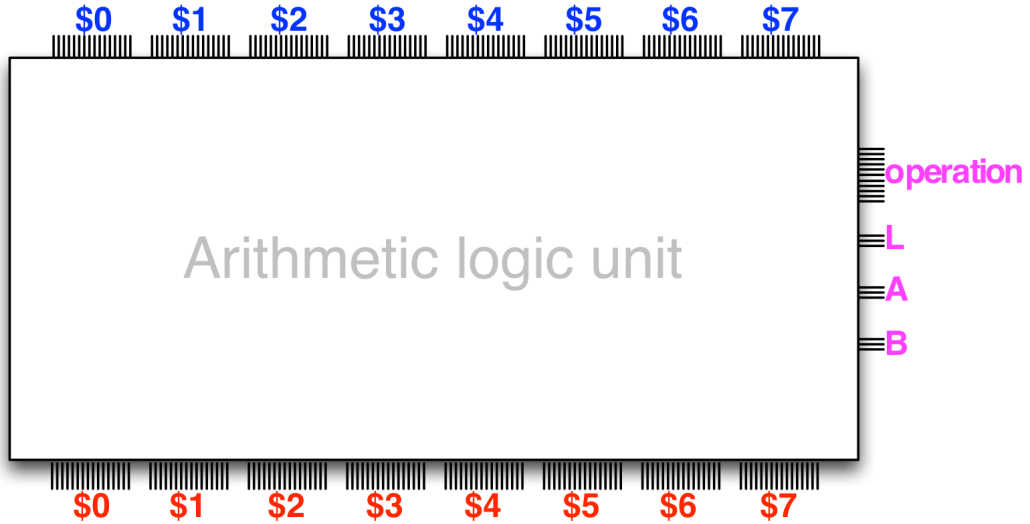
Rekisterit (registers)

- **Rekisteri toimii kuin piiriin sisäänrakennettu ‘muuttuja’**
- **Meillä oli esimerkki piiristä, joka pystyy pitämään ja päivittämään tilaansa**
  - Akkumulaattori-summaimessa on sisäinen tila, joka päivittyy
- **Prosessoriarkkitethuurissa tällaisia muuttujia kutsutaan rekistereiksi (registers)**
- **Meidän arkkitehtuurissa on kahdeksan 16-bittistä rekisteriä:**
  - `$0`,`$1`,`$2`,`$3`,`$4`,`$5`,`$6` ja `$7`





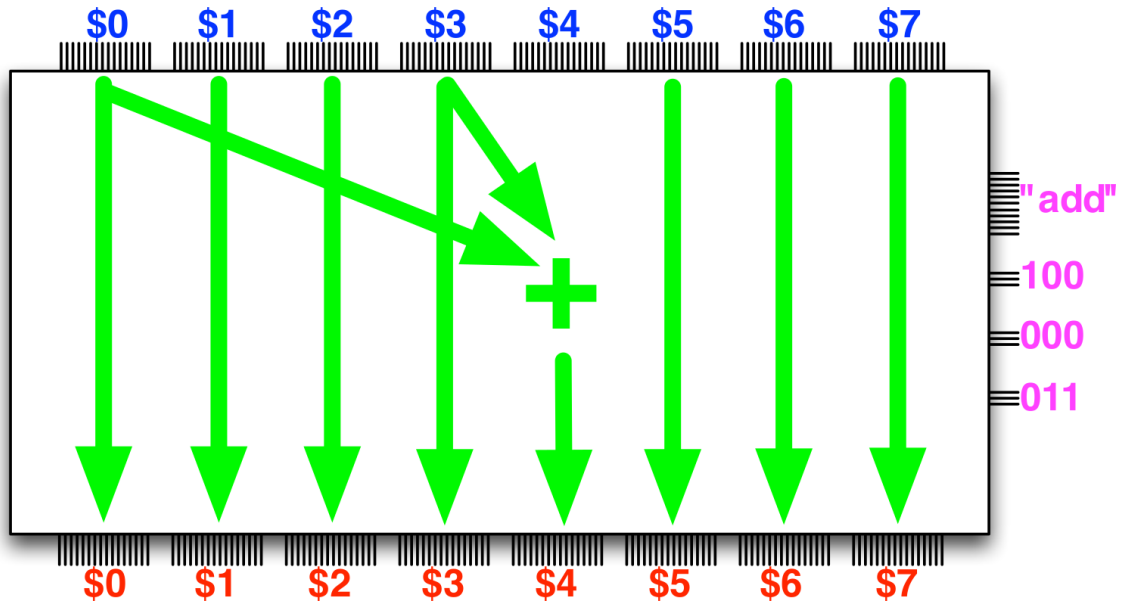
Aritmeettisloogginen yksikkö (arithmetic logic unit, ALU)



- **Sama periaate suunnittelussa kuin akkumulaattori-summain-vähentimessä**
- **Kahdeksan rekisteriä: `$0`–`$7`**
- **Toiminnot kuten lisäys, vähennys, siirto, AND, OR, …**
  - `operation` kertoo valitun toiminnon (operaatio, operation)
- **`L` on rekisterin numero (0-7), mihin tulos tallennetaan**
- **`A` on rekisterin numero (0-7) ensimmäiselle operandille**
- **`B` on rekisterin numero (0-7) toiselle operandille**

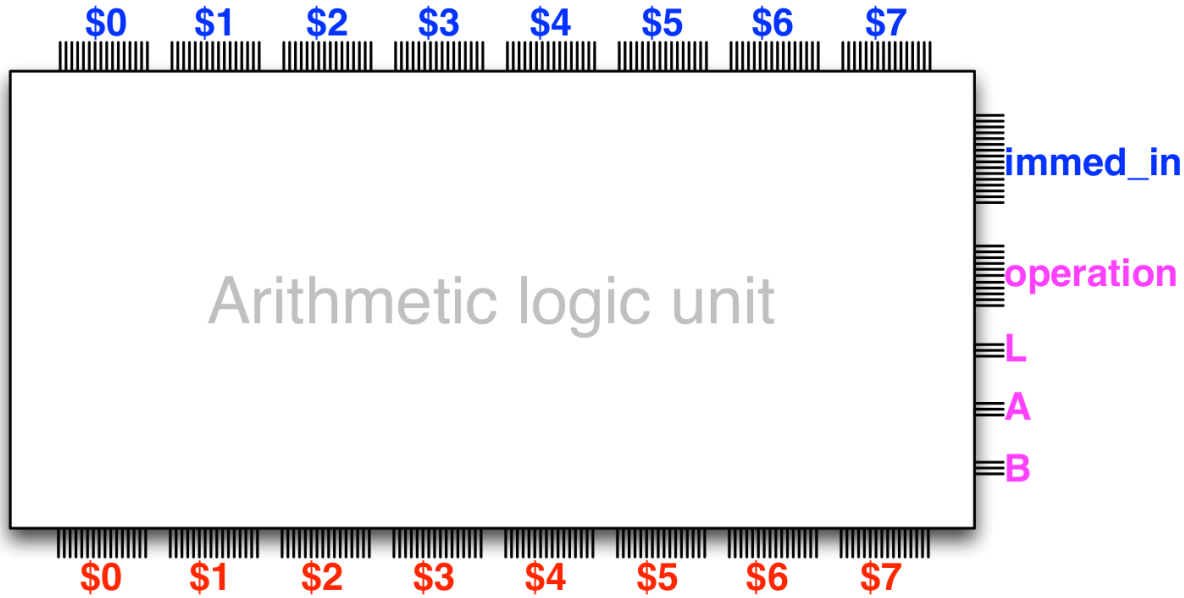
Aritmeettisloogginen yksikkö

- **Esim: jotta ALU laskisi `$0`:n ja `$3`:n summan ja laittaisi sen rekisteriin `$4`**
  - Aseta `operation` numeroksi, joka vastaa yhteenlaskukäskyä
  - Aseta `L` olemaan `100` (4)
  - Aseta `A` olemaan `000` (0)
  - Aseta `B` olemaan `011` (3)
  - Kellota piiri



Välittömän datan asettaminen ALUun

- **Pienellä päivityksellä ALU voi käsitellä ‘vakioita’ operaatioissa**
- **`immed_in` on 16-bittinen sana jota voidaan käyttää rekisterin sijaan**
  - Ei tarvi hukata rekisteriä pitämään vakiota

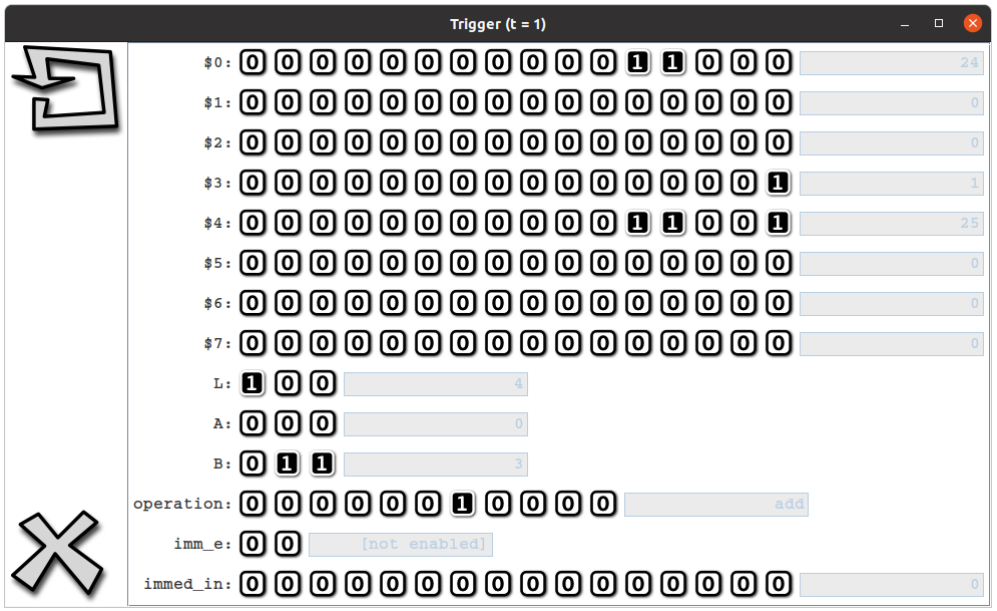


- **Kaksi versiota joillekin käskyille (instructions), esim. add:**
  - Laske yhteen kaksi rekisteriä, jotka osoitetaan `A`:ssa ja `B`:ssä
  - Laske yhteen `A`:n osoittama rekisteri ja vakio, joka on tallennettu `immed_in`:iin

ALU:n testailu

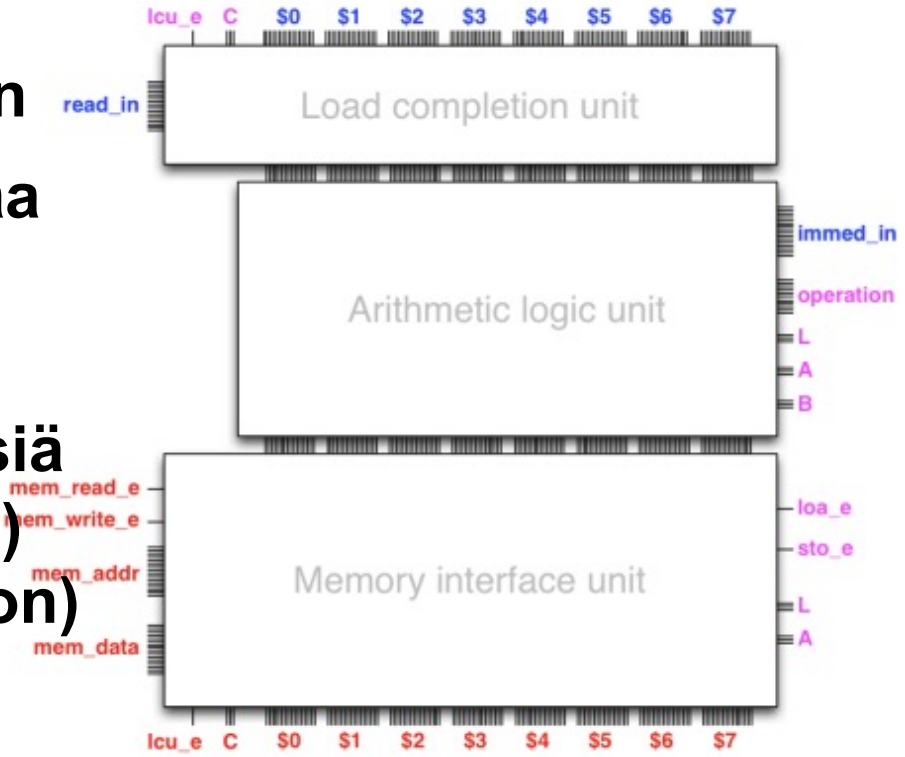
- **Harjoitusten `armlet`-paketista löytyy käyttöliittymä, jonka avulla voit kokeilla ALUa asettamalla eri syötteitä ja kellottamalla piiriä**
- **Aja ohjelma `launchALUTrigger.scala` tai**
- **Luo ALU trigger konsolissa (kun olet importoinut minilogin)**

```
scala> new ALUTrigger()
```



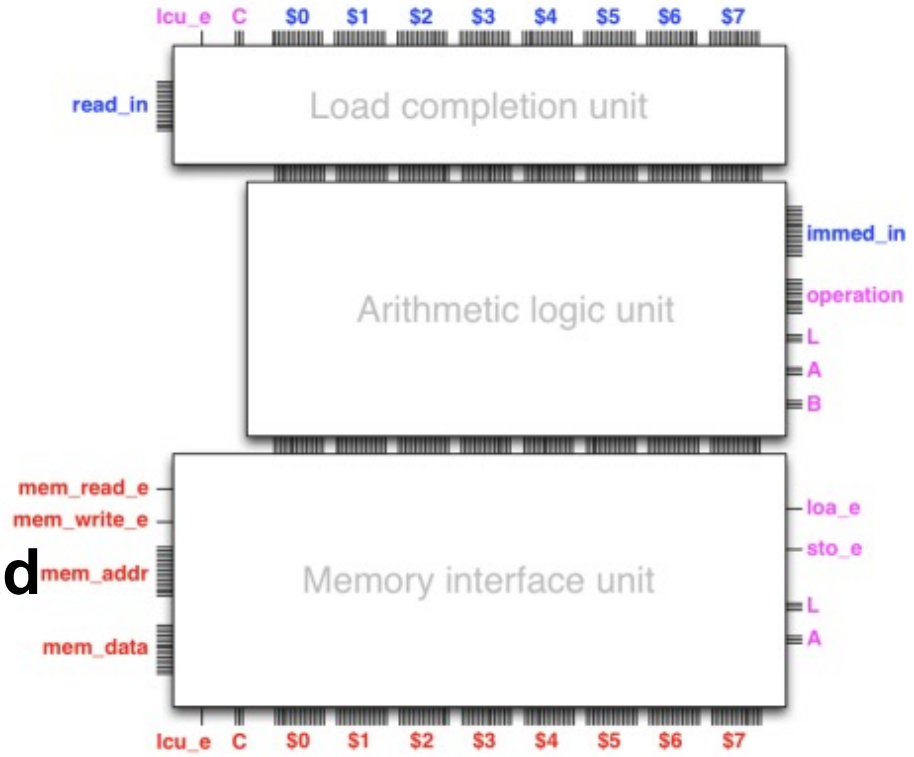
Datan tallentaminen (storing) ja lataaminen (loading)

- **Kahdeksan 16-bittistä rekisteriä tuskin riittää useimpiin toimintoihin**
- **Suurempi tallennus vaatii rajapintaa (prosessorin) ulkopuolisiin muistipiireihin (memory circuits)**
- **Meidän prosessori käyttää 16-bittisiä sanoja, eli se voi osoittaa (address) 2^{16} muistipaikkaan (memory location)**
  - Jokainen muistipaikka voi sisältää yhden 16-bittisen datasanan (2 tavua)
  - Eli 2^{16} * 2 tavua = 128 kibitavua tietoa



Datan tallentaminen ja lataaminen

- **Muistin rajapintayksikkö (memory interface unit) on vastuussa muistitoiminnoista**
  - Muistista lataaminen (load from memory)
  - Muistiin tallentaminen (store to memory)
- **Latauksen viimeistely-yksikkö (load completion unit) on vastuussa muistista haetun arvon asettamisesta rekisteriin tietopolun alussa**



Joko meillä on ohjelmoitava tietokone?

- **Melkein!**
- **Tarvitsemme vielä tavan ladata ohjelmia – tietopolun ohjeita (the instructions for the data path) – ja suorittaa ne automaattisesti**
  - Tästä seuraavalla kerralla

Harjoituksissa tällä viikolla

- **Takaisinkytkentä (feedback)**
- **Oskillaattori (oscillator)**
- **Lineaariset takaisinkytketyt siirtorekisterit (linear-feedback shift registers)**
- **Muisti**
  - Yhden sanan muistimoduuli
  - Useamman sanan muistimoduuli – tästä videoissa vinkki
- **Sekventiaalinen kertolaskin (sequential multiplier)**
  - Käytä yksinkertaisen summaimen toimintoa toistaen
- **(Haastetehtävä: putkitettu kertolaskin (pipelined multiplier)**

Harjoituksissa tällä viikolla

- **Huom. minilogissa XOR:lle toimii sekä ^^ että +, ne tekee saman**
- **Vinkkejä:**
  - Saat aina `Gate`sta tai `Bus`ista `Circuit`in käyttämällä `.host`-metodia
  - **Voit kutsua `buildFeedbackFrom`-metodia vain `inputElement`sille**
  - Memory-tehtävä: huomaa annetut funktiot `buildDecoder` ja kaksi funktiota `*Selector`!
  - Multiplier-tehtävä: huomaa `loadEnable`:n ja `ready`:n kuvaukset
  - Muista että `Bus` on sekvenssi!
- **Tarkista ohjelmasi toiminnallisuus `play`-ohjelmien avulla**
  - **Huomaa, että sinun pitää poistaa kommentteja, jotta osa koodista toimisi!** (koska koodissa visualisoidaan eri tapauksia)
  - Älä unohda viimeistä `t.go()` -riviä