CS-A1120 Ohjelmointi 2

Aalto-yliopisto
Perustieteiden
korkeakoulu

Luento 5
kevät 2026

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

23.3.2026

https://presemo.aalto.fi/o2fi2026

Tänään O2:ssa

https://presemo.aalto.fi/o2fi2026

Ohjelmoitava tietokone

Tentti-ilmo auki
Sisussa 14.5. asti

ILMOITTAUDU!

Aalto-yliopisto
Perustieteiden korkeakoulu

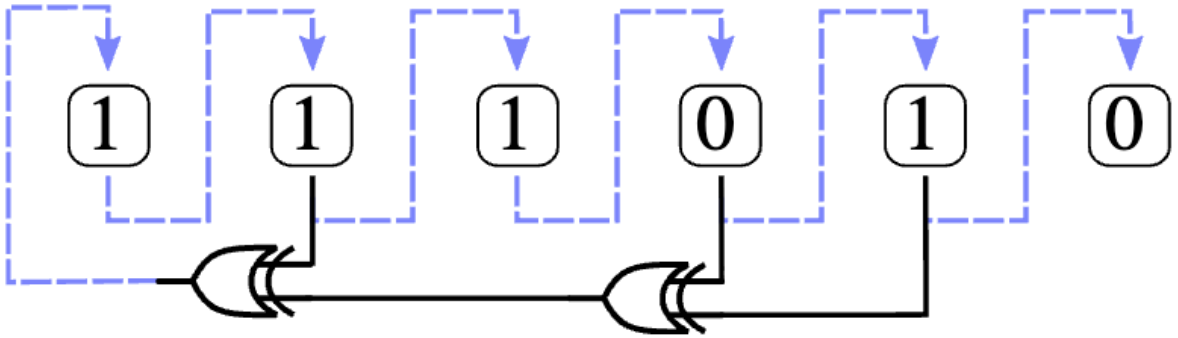
Kierroksen 5 oppimistavoitteet

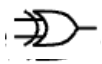
- **Tämän viikon jälkeen,**
  - Osaat selittää tallennettuja ohjelmia käyttävien tietokoneiden ja automaattisen suorituksen arkkitehtuuriperiaatteet
  - Tunnet assembly-kielien roolin
  - Osaat toteuttaa yksinkertaisia algoritmeja kurssin `armlet`-assembly –kielellä
  - Olet tietoinen laskennallisesta yleismaailmallisuudesta

Kertaus takaisinkytkennästä - kysymys

https://presemo.aalto.fi/o2fi2026

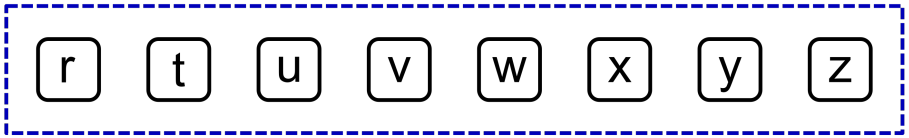
- **Pohdi seuraavaa sekvenssilogiikan piiriä:**



- **Jossa**  **operaattori XOR (poissulkeva-tai)**
- **Aluksi (ajan hetkellä *t* = 0) bitit ovat, vasemmalta oikealle, kuten kuvassa `11101`**
- **Mitä ovat bittien arvot (vasemmalta oikealle), kun piiriä on kellotettu yhden kerran, eli *t* = 1?**

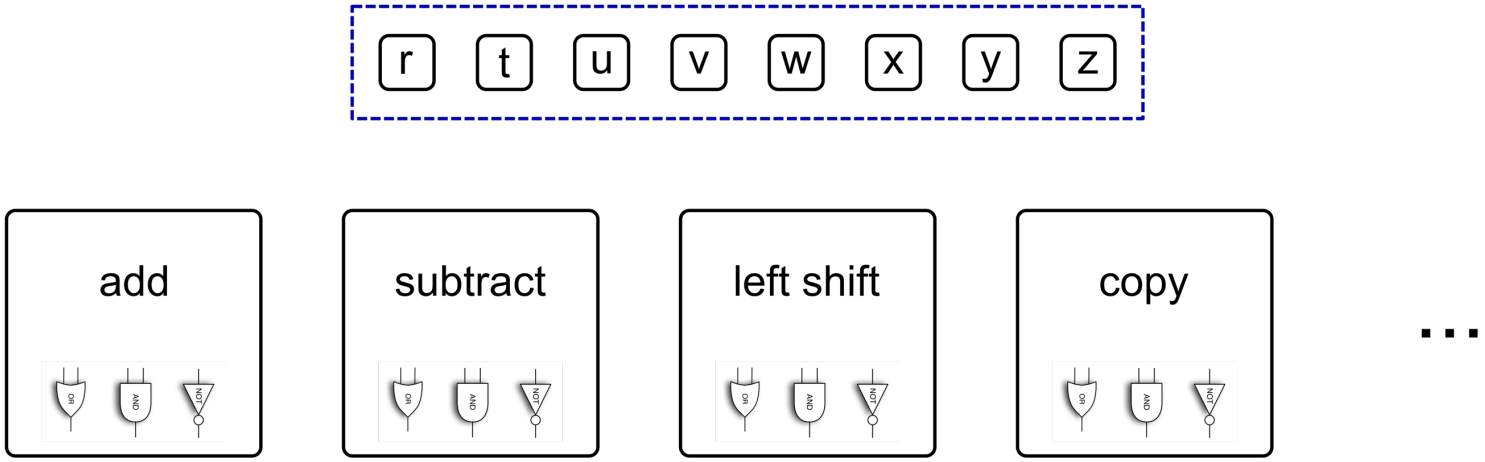
Ohjelmoitava kone, tähän mennessä

- **Datan binääriesitys (tässä 8-bittinen väylä)**



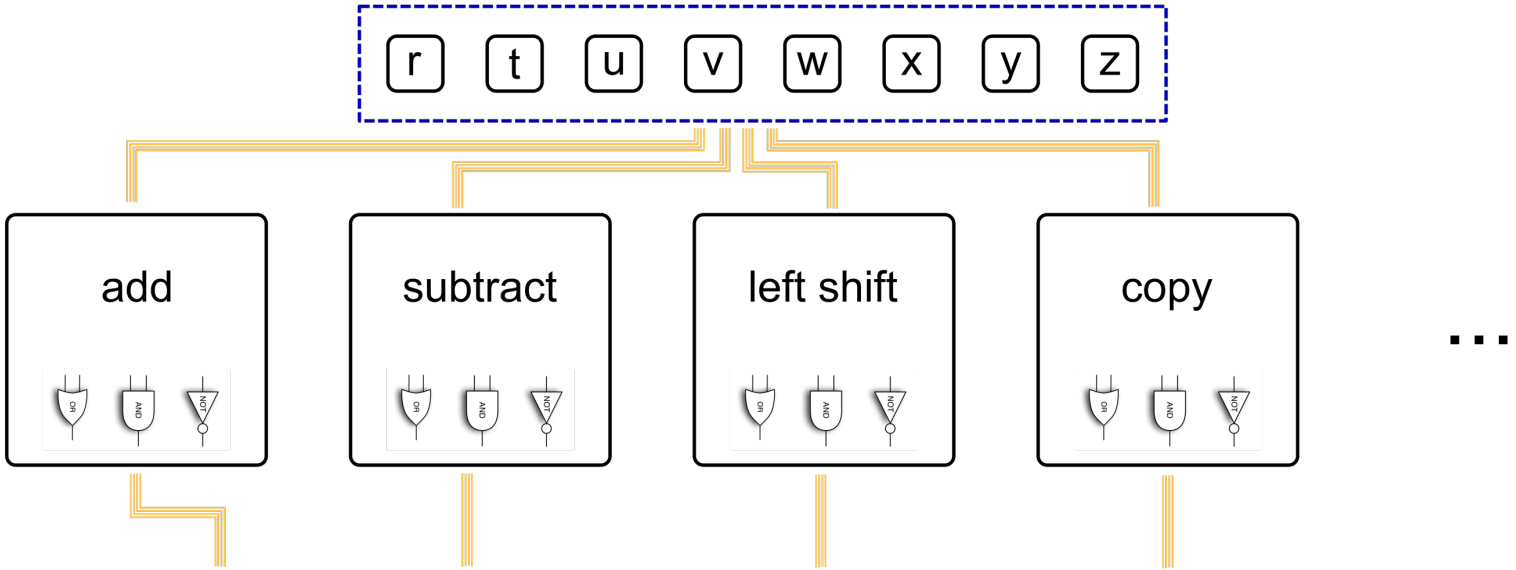
Ohjelmoitava kone, tähän mennessä

- **Kombinaatiologiikka**



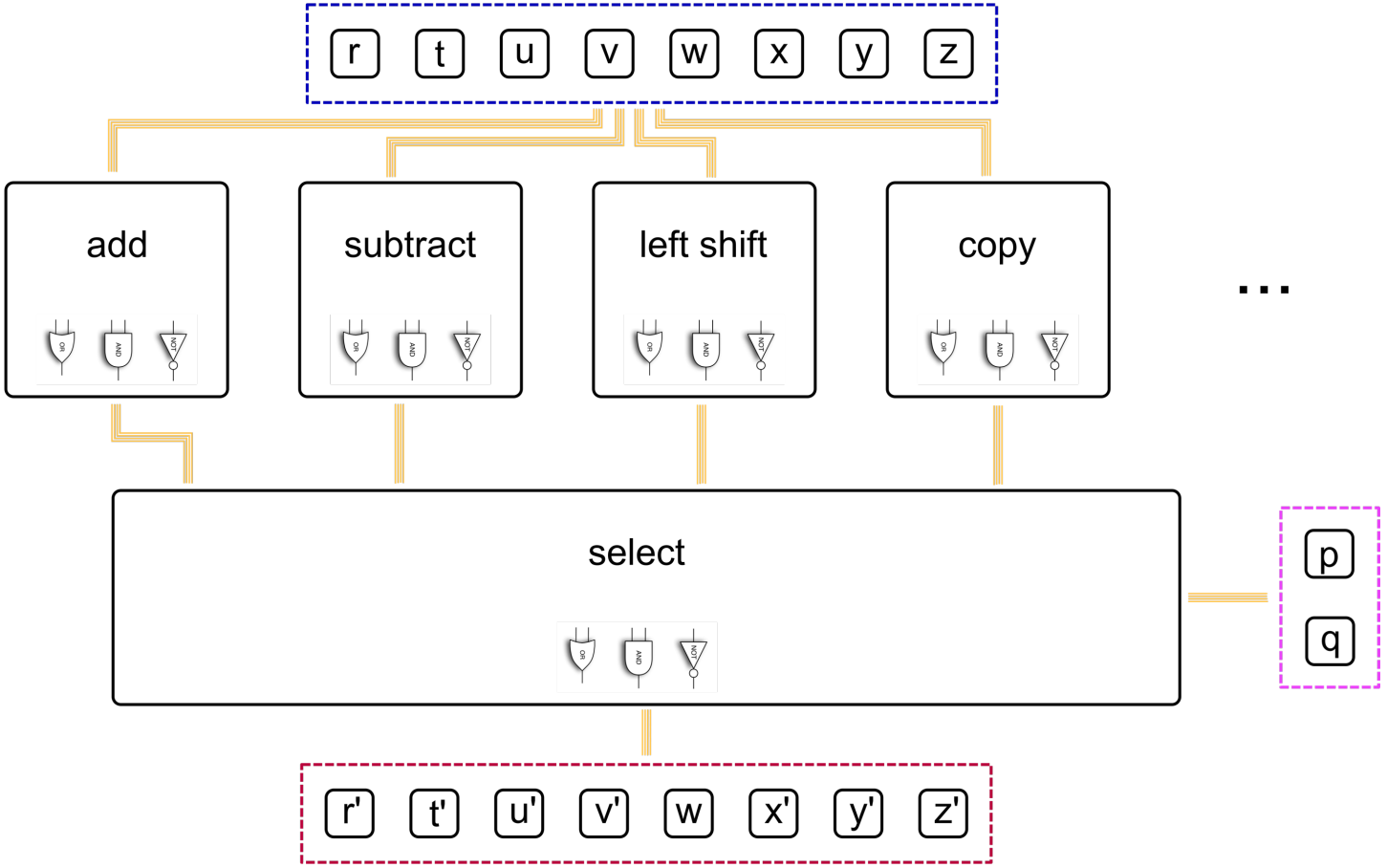
Ohjelmoitava kone, tähän mennessä

- **Kombinaatiologiikka**



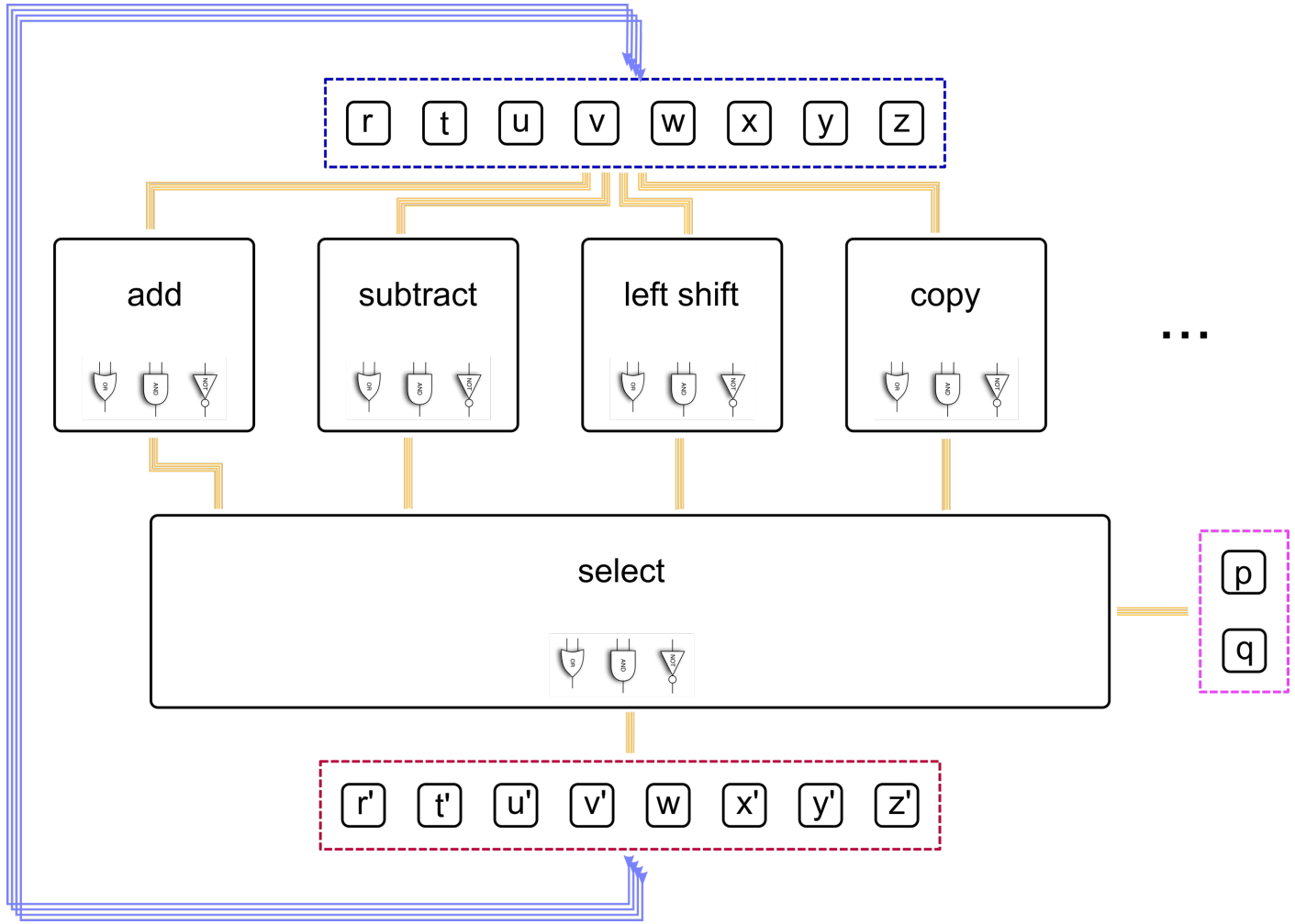
Ohjelmoitava kone, tähän mennessä

- **Kombinaatiologiikka**



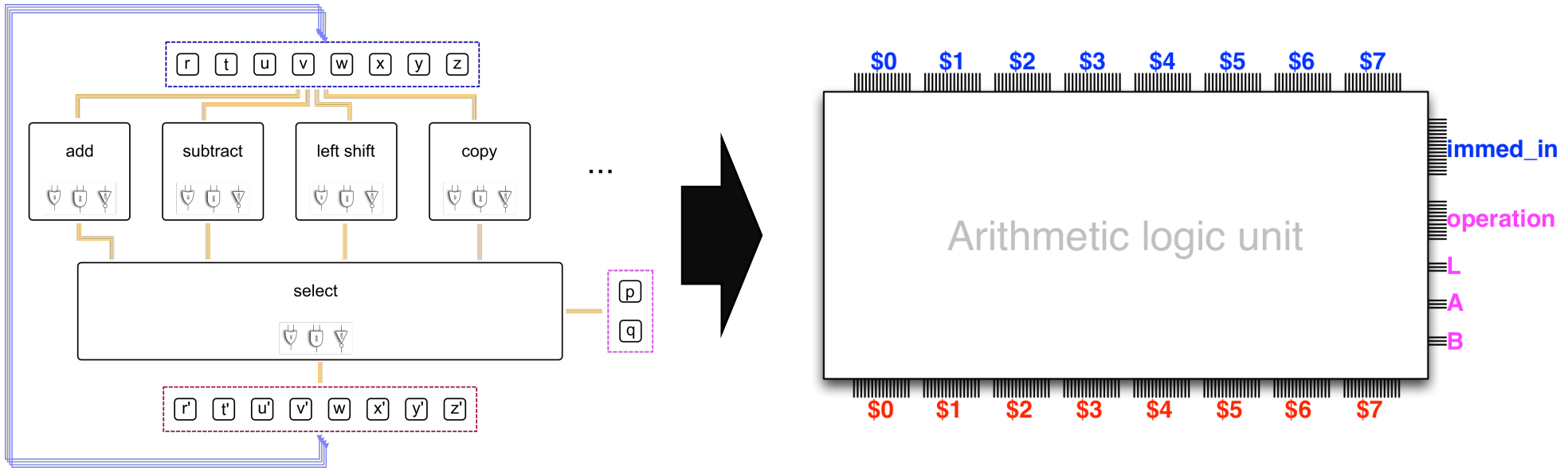
Ohjelmoitava kone, tähän mennessä

- **Sekvenssilogiikka tilatiedon säilyttämiseksi (tila = state)**



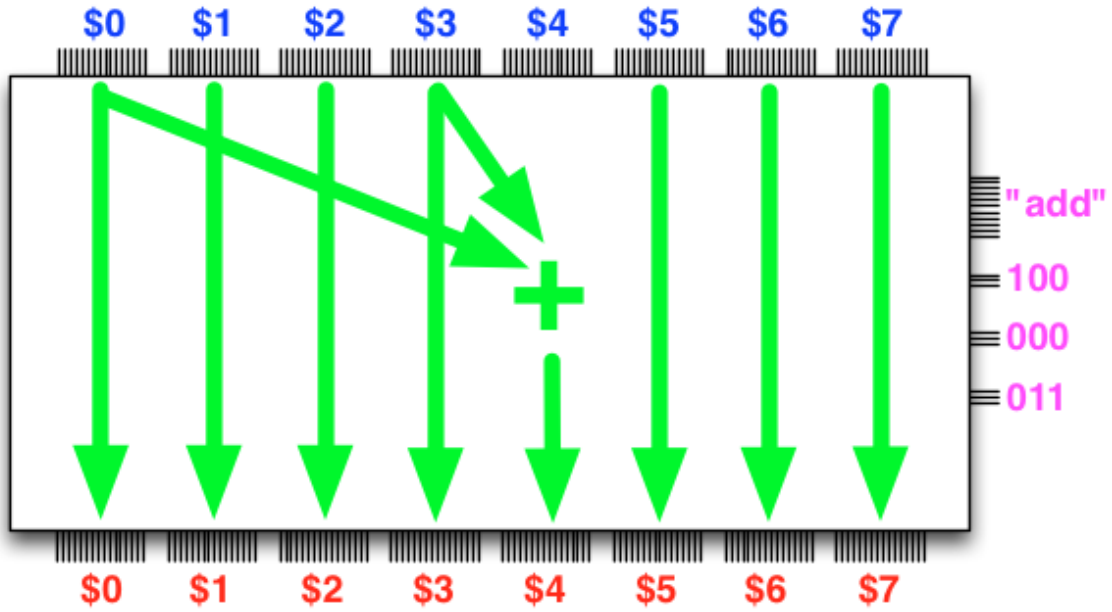
Näillä saamme tehtyä aritmeettisloogisen yksikön (Arithmetic Logic Unit, ALU)

- **ALU huolehtii laskennasta ja logiikasta tietokoneen suorittimessa**



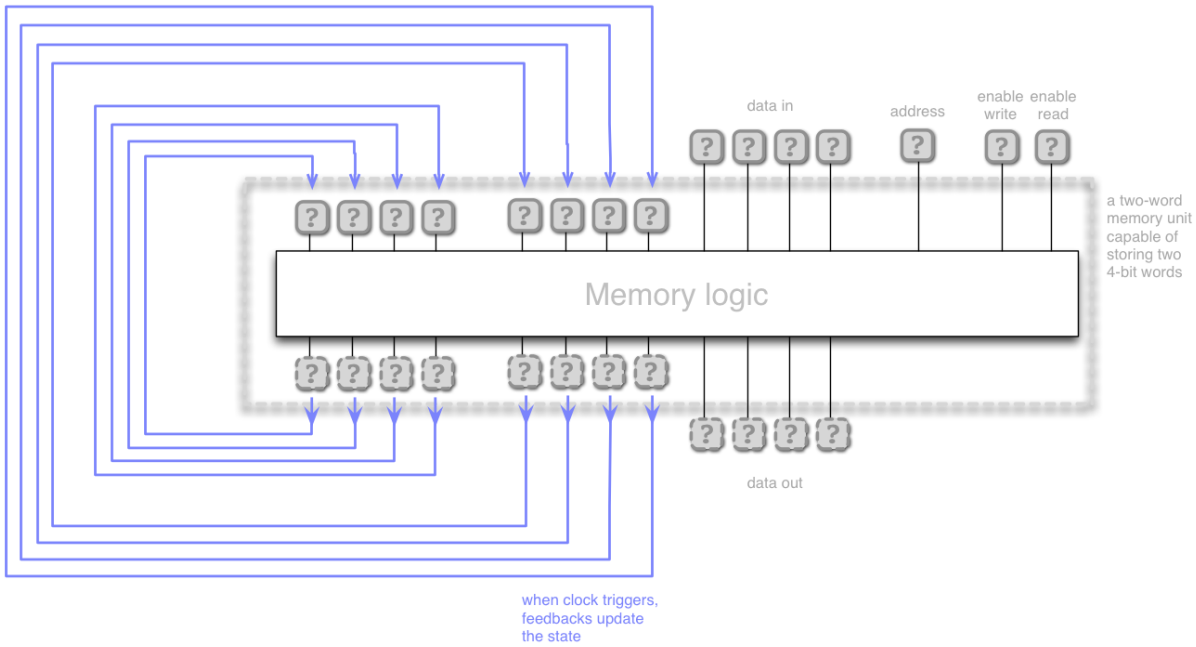
Armlet-arkkitehtuurin ALU, tähän mennessä

- **$0-$7 ovat rekistereitä (registers), jotka pitävät yllä ALUn sisäistä tilaa**
- **Datan syöttö sisään `immed_in` (sisään syötettävä vakio)**
- **`operation` määrittää, mikä toiminto suoritetaan**



- **`A` ja `B` kertoo, mitkä on syöterekisterit**
- **`L` kertoo, mihin rekisteriin lopputulos laitetaan**

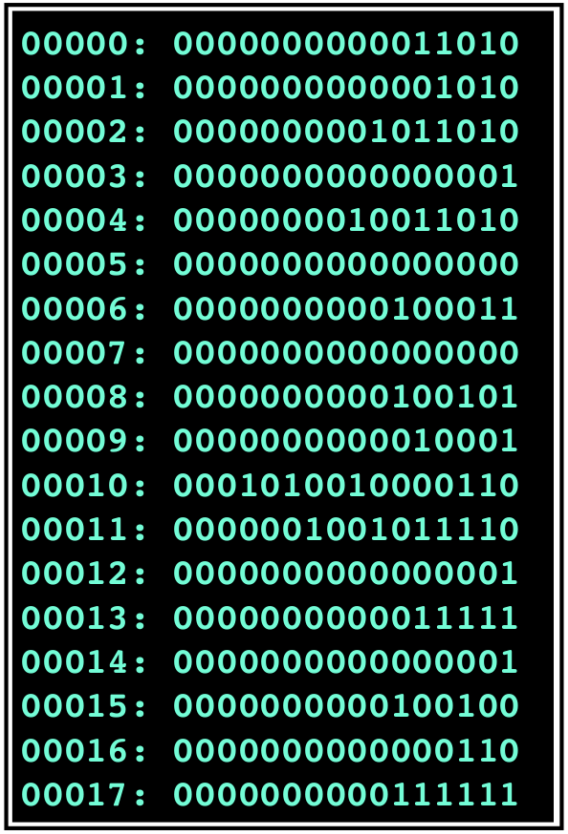
Tähän mennessä: muisti



- **Sekvenssilogiikka auttaa säilyttämään tilan eli rakentamaan muistin**
- **Sisäisesti jokainen muistin sana säilyttää tilansa (esim. käyttäen takaisinkytkentää), kunnes se päivitetään päällekirjoittamalla**

Tähän mennessä: muisti

- **Muistimoduuli on piiri useammasta yksisanaisesta muistista, joihin voidaan osoittaa (addressed, indexed)**
  - Muistia voidaan ajatella tauluna
- **Muistiosoitteen väylän leveys (width) (bittien/porttien määrä) ja i/o-väylän sanan koko lopulta määrittelee muistin kapasiteetin**
- **Muisti, jossa `mem_addr`:n koko on `n`, voi sisältää 2^n sanaa dataa**

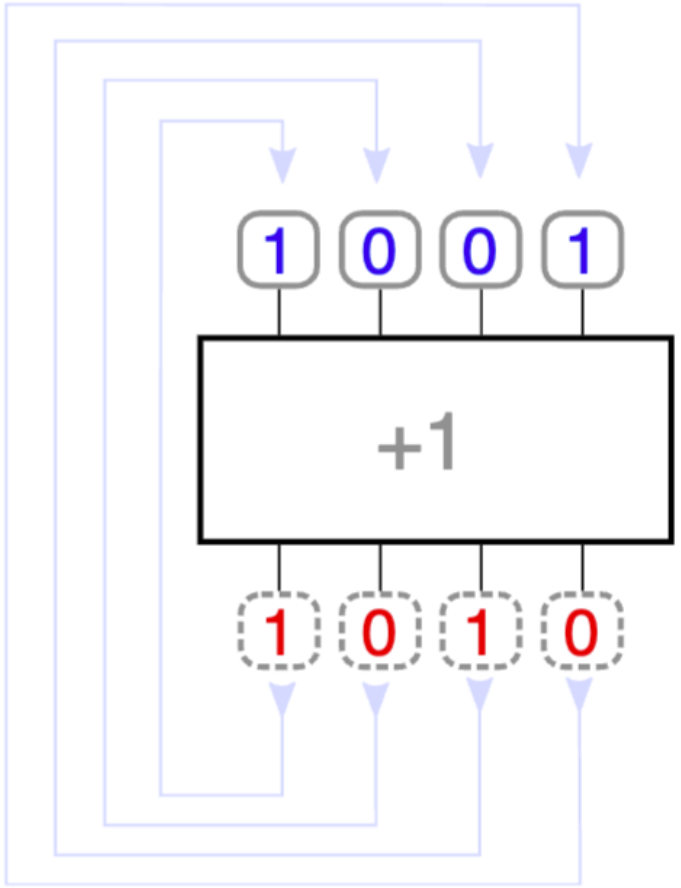


Tähän mennessä: muisti (`armletissa`)

- **Muistimoduulin konsepti (`armlet`)**
  - `mem_addr` on muistin osoiteväylä (memory address bus)
  - Portit `mem_read_e` ja `mem_write_e` määrittävät kirjoittamista/lukemista osoitteesta
  - `mem_data` ja `read_out` ovat muistin syöte- ja ulostuloväyliä (input/output busses)

Tähän mennessä: kello (ja laskuri)

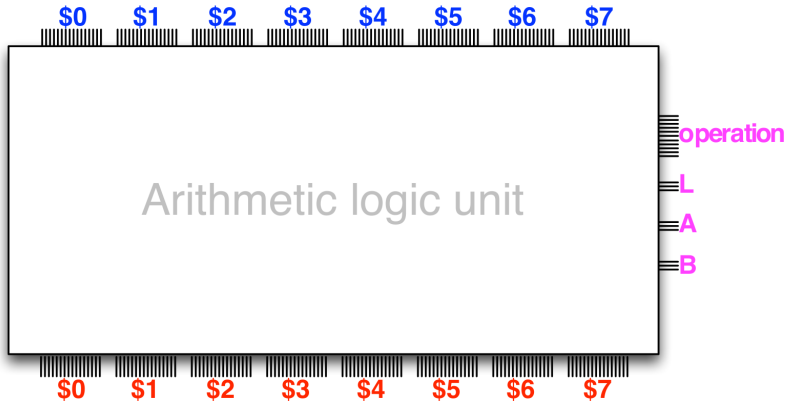
- **Kello (clock) laukaisee takaisinkytkennän piirissä säännöllisesti**
  - Nykyiset prosessorit toimivat GHz-taajuusalueella
- **Kellopulssi on kuin syke, joka ajaa piirin sekvenssilogiikkaa**
  - Seuraava käsky jonosta,
  - Vai jonkun muun käskyn?



Ohjelma

Aalto-yliopisto
Perustieteiden
korkeakoulu

Millaisia ohjelmia kirjoitetaan ALUlle?



- **Ohjelma on sekvenssi syötteitä `operation, L, A, B` (yhteensä 16 bittiä = 1 sana) ja tarvittaessa `immed_in` (16 bittiä)**
- **Kuinka ne voidaan tallentaa?**
  - Ne voisi kirjoittaa paperille ja
  - Sitten syöttää yksi kerrallaan ALUlle kääntelemällä rekisterien syötebittejä…
  - Ja sitten kellottaa ALUa ja toistaa seuraavalle komennolla

  Tylsää….(ja hidasta)
- **Kehitelläänpä kone, joka lukee automaattiesesti seuraavan komennon (instruction)**

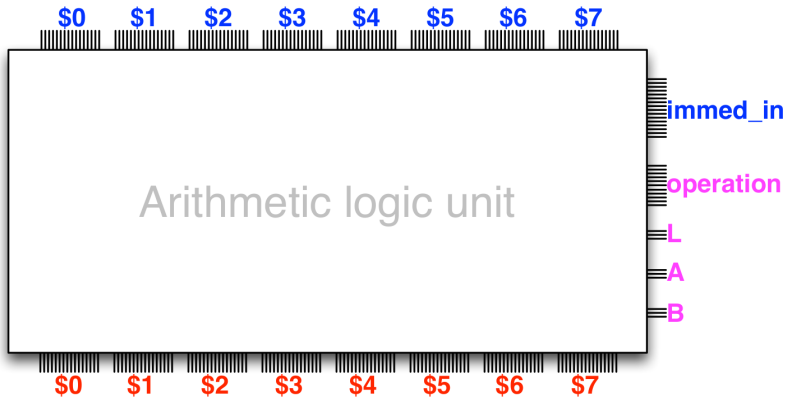
Ohjelma

- **Jotain, jonka avulla konetta käsketään siirtymään tilasta toiseen**
- **Kone suorittaa (execute) ohjelmaa automaattisesti, yksi käsky kerrallaan**
- **Konekieli (machine code)**
- **Oikealla muistia:**
  - Ensin muistiosoite desimaalilukuna
  - Sitten merkki :
  - Lopuksi käsky binäärinä (ilman dataa tai datan kanssa)

```
00000: 0000000000011010
00001: 0000000000001010
00002: 0000000001011010
00003: 0000000000000001
00004: 0000000010011010
00005: 0000000000000000
00006: 0000000000100011
00007: 0000000000000000
00008: 0000000000100101
00009: 0000000000010001
00010: 0001010010000110
00011: 0000001001011110
00012: 0000000000000001
00013: 0000000000011111
00014: 0000000000000001
00015: 0000000000100100
00016: 0000000000000110
00017: 0000000000111111
```

Laitetaan ohjelma muistiin!

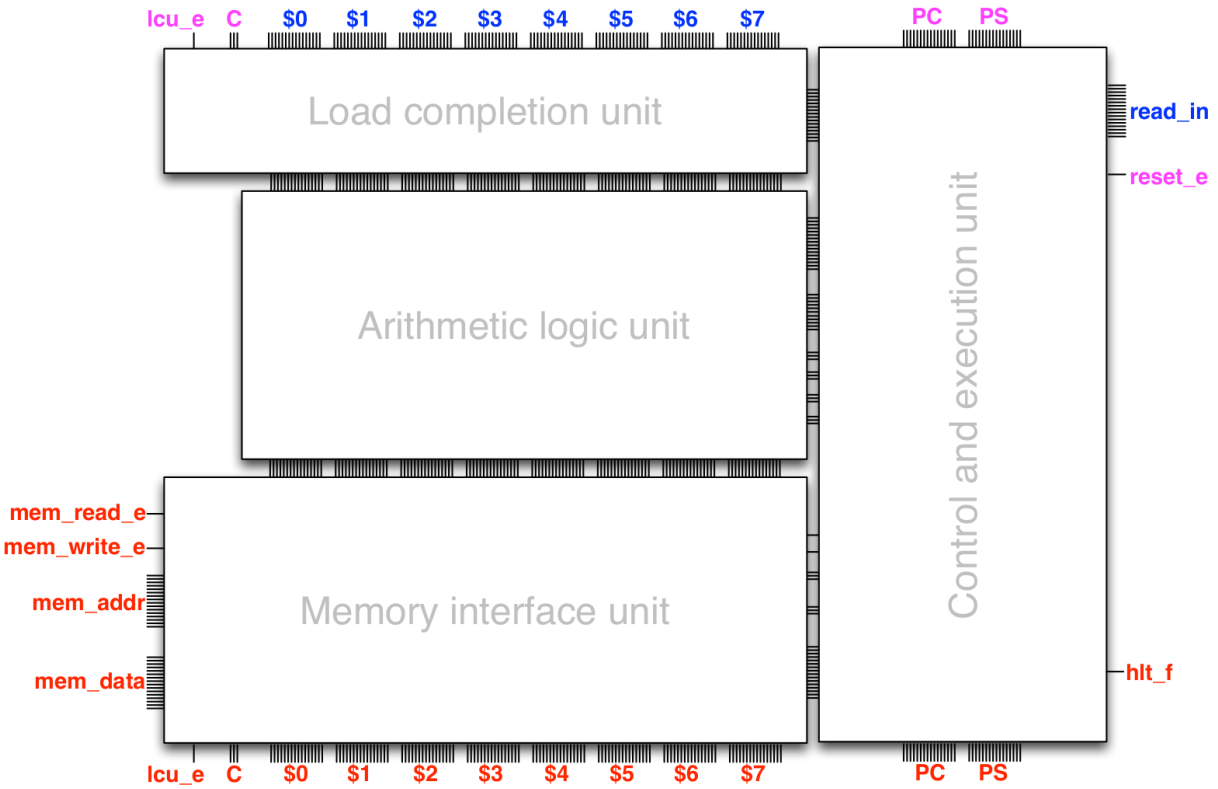
- **Eli ohjelma (program) on sekvenssi käskyjä (bittejä) muistissa**
  - Ohjelmalaskuri (program counter) sisältää tällä hetkellä suoritettavan käskyn muistiosoitteen
  - Jokaisella kellonlyömällä (tick) käsky ladataan muistista ALUun
- **Tarvimme siis vielä yhden palikan…**




```
00000: 0000000000011010
00001: 0000000000001010
00002: 0000000001011010
00003: 0000000000000001
00004: 0000000010011010
00005: 0000000000000000
00006: 0000000000100011
00007: 0000000000000000
00008: 0000000000100101
00009: 0000000000010001
00010: 0001010010000110
00011: 0000001001011110
00012: 0000000000000001
00013: 0000000000011111
00014: 0000000000000001
00015: 0000000000100100
00016: 0000000000000110
00017: 0000000000111111
```

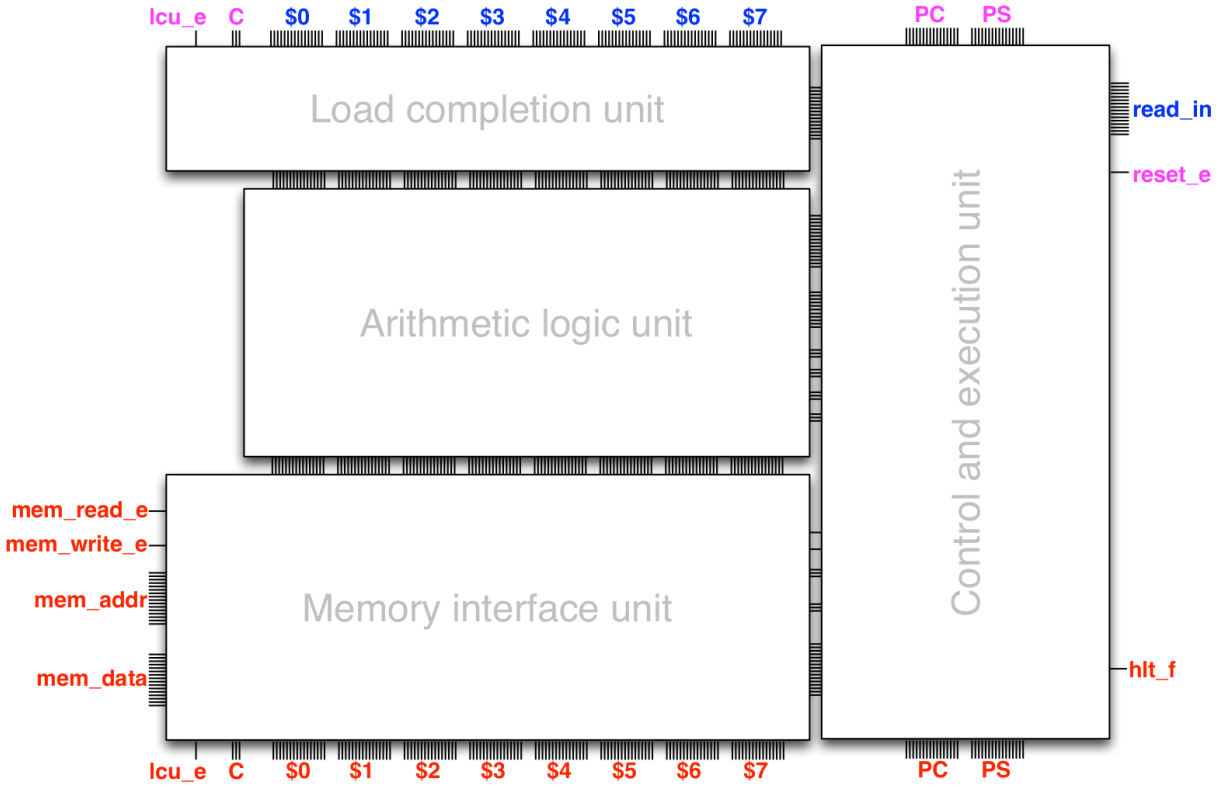
Armlet – ohjaus- ja suoritusyksikkö (control & execution unit)

- **Ohjelmalaskuri (program counter, PC) on erityinen rekisteri, joka kertoo mistä muistiosoitteesta seuraava käsky ladataan**
  - Oletus: $PC_{t+1} = PC_t + 1$
- **Oletuksena ohjelma tekee yhden käskyn kerrallaan ja etenee järjestyksessä**
  - Mutta voidaan myös hypätä (jump) tai haarautua (branch)



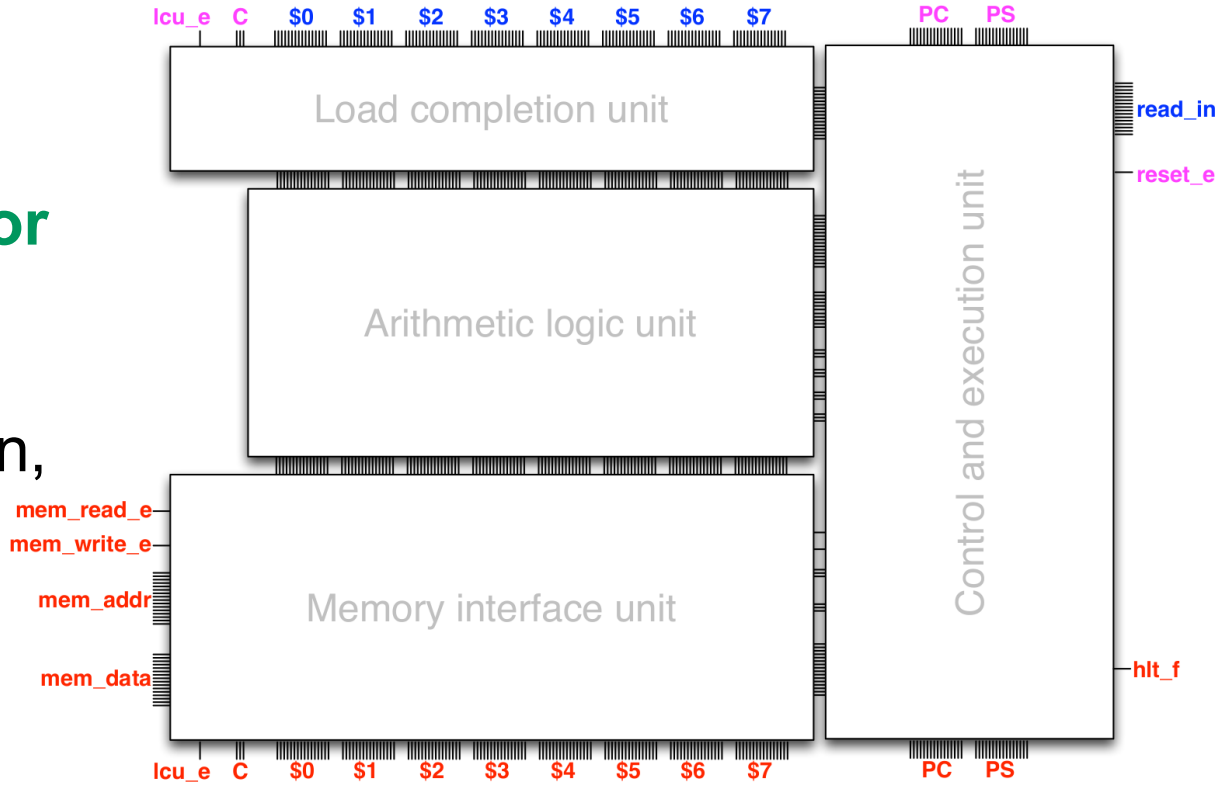
Hyppy (jump)

- **Oletusjärjestystä voidaan muuttaa käskyillä**
  - jump: PC_{t+1} = <osoite>



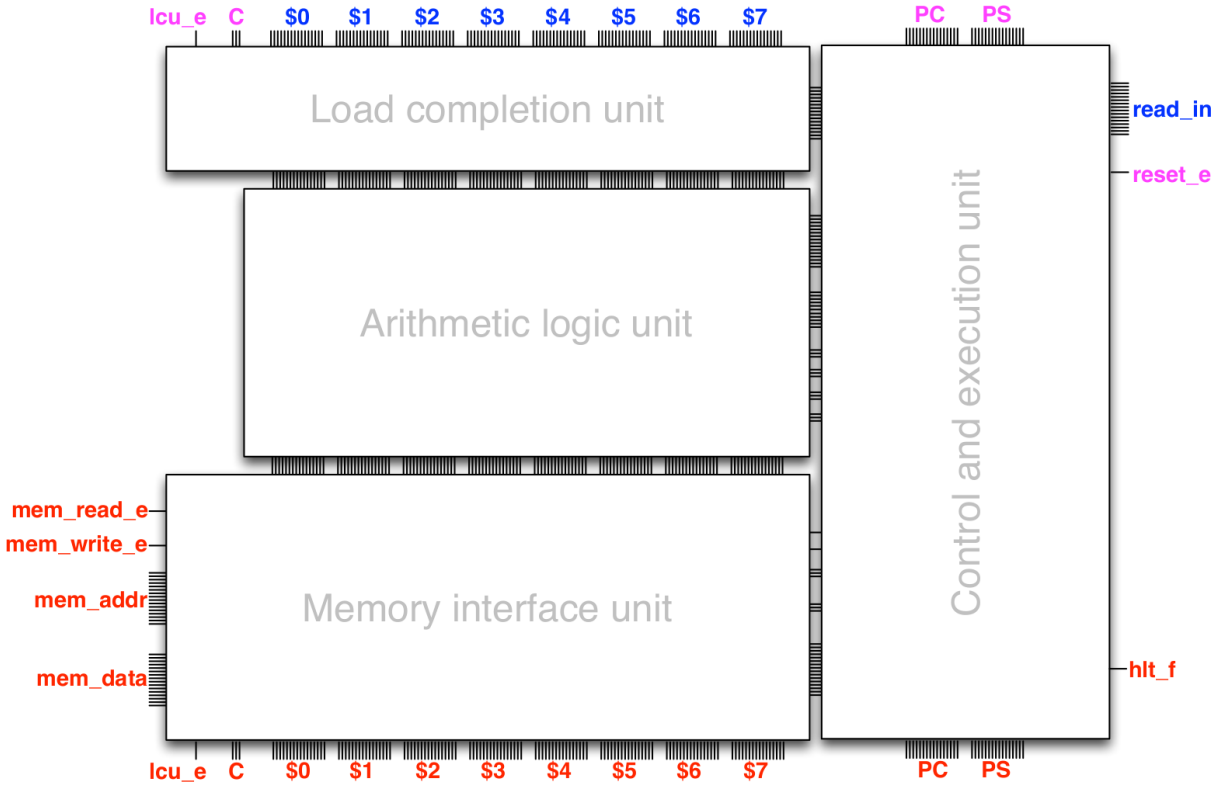
Ehdollinen haarautuminen (conditional branching)

- **Välillä pitäisi valita vaihtoehtojen välillä**
- **Prosessorin tila (processor status, PS) rekisteri**
  - Rekisterin arvo viittaa edellisen operaation tilaan, esim. vertailuoperaatiot
  - branch PC_{t+1} = <osoite>, jos jokin ehto pätee, muuten PC_{t+1}



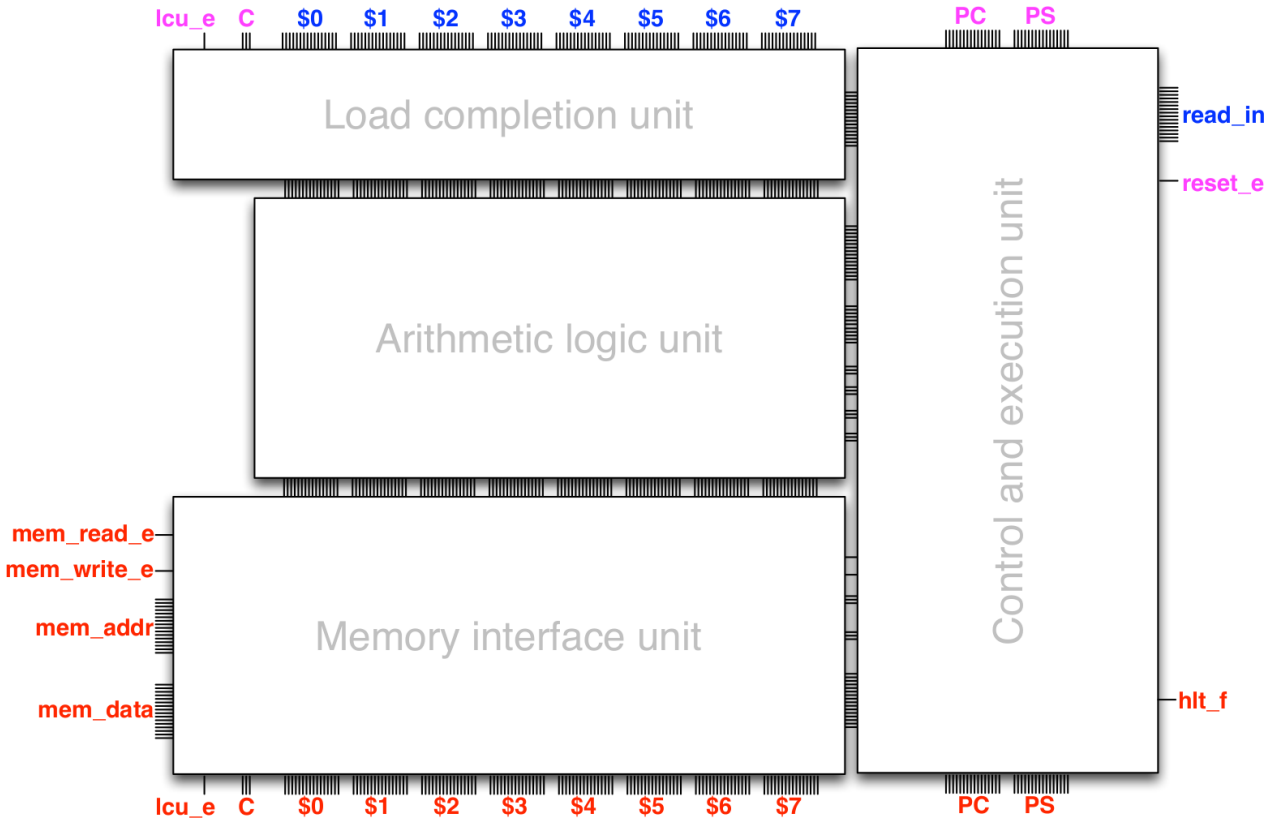
Ehdollinen haarautuminen (conditional branching)

- **Yhdessä PC ja PS mahdollistavat siis silmukat ja ehdollisen suorituksen!**



Armletin arkkitehtuuri

- **Armlet on esimerkki, jossa käytetään von Neumann -arkkitehtuuria (Princeton-arkkitehtuuri)**
  - Käskyjen hakeminen ja datan operointi ei voi tapahtua samaan aikaan, koska ne jakavat saman syöteväylän



Pysähtyminen (halting) ja keskeyttäminen (trapping)

- **Kuinka prosessori tietää, milloin sen pitää lopettaa suorittaminen?**
  - Tavallisesti prosessori jatkaa käskyjonon lukemista eteenpäin

- **Erityinen pysäytyskäsky: halt (halting)**
  - Suorituksen pysäyttäminen (ilman mahdollisuutta jatkaa)
  - Eteepäin pääseminen edellyttää uudelleenkäynnistämistä (reset)
- **Käsky keskeyttämiselle (trap, trapping)**
  - Suorituksen pysäyttäminen väliaikaisesti
  - Näppärää debuggaukselle

Ohjelmointi

- **Periaatteessa voimme ohjelmoida tietokonetta asettamalla haluamiamme käskyjä vastaavat bitit muistiin**
- **Syöte on bittejä: kaikki käskyt (ja data) on numeroita: konekoodi (machine code)**
  - Ensimmäisiä tietokoneita ohjelmoitiin asettamalla näitä numeroita suoraan (esim. reikäkorttien avulla)
  - Todella vaikea ihmiselle seurata, mitä tapahtuu

Armlet Machine code(s)	Armlet assembly	High-level language
`0111000010000111`	`sub $2, $0, $7`	r2 = r0 - r7
`0000001011011100 0011000000111001`	`ior $3, $1, 12345`	r3 = r1 | 12345

Ohjelmointi: Assembly

- **Assembly-kieli on yhden-suhde-yhteen käännös näistä numero-käskyistä (hieman) ihmisluettavammassa muodossa**
- **Kullakin prosessoriarkkitehtuuriperheellä on oma assembly-murre**
- **Ensimmäinen Assembly-kieli oli Kathleen Boothin kirjoittama**
- **Assembly-kieli on symbolinen konekieli**

Armlet Machine code(s)	Armlet assembly	High-level language
`0111000010000111`	`sub $2, $0, $7`	r2 = r0 - r7
`0000001011011100 0011000000111001`	`ior $3, $1, 12345`	r3 = r1 | 12345

Konekielen kääntäjä (Assembler)

Symbolinen armlet-konekielinen ohjelma

Symbolic `armlet` machine language program

```
        mov $0, 10
        mov $1, 1
        mov $2, 0
@loop:
        cmp $0, 0
        beq >done
        add $2, $2, $1
        add $1, $1, 1
        sub $0, $0, 1
        jmp >loop
@done:
        hlt
```

Konekielen kääntäjä

Machine code compiler

= Assembler

Machine language program (`armlet`-binary)

```
00000: 0000000000011010
00001: 0000000000001010
00002: 0000000001011010
00003: 0000000000000001
00004: 0000000010011010
00005: 0000000000000000
00006: 0000000000100011
00007: 0000000000000000
00008: 0000000000100101
00009: 0000000000010001
00010: 0001010010000110
00011: 0000001001011110
00012: 0000000000000001
00013: 0000000000011111
00014: 0000000000000001
00015: 0000000000100100
00016: 0000000000000110
00017: 0000000000111111
```

(Loadable/executable) `armlet` binary

Konekielinen ohjelma (armlet-binääri)

Assembly-kielinen ohjelma

Assembly-language program

Ladattava / suoritettava armlet-binääri

Tietokoneen salaisuus

Aalto-yliopisto
Perustieteiden
korkeakoulu

Tietokoneen salaisuus – onko se nyt ratkaistu?

- **Kyllä, eräässä mielessä**
  - Voimme (konseptuaalisesti) rakentaa ohjelmoitavan koneen
  - Jos voimme toteuttaa sekvenssilogiikan rakennuspalikat
  - Tämä voidaan tehdä laitteiston (hardware) avulla
  - Ohjelma on jono binäärikäskyjä tässä arkkitehtuurissa
  - Ihmiset yleensä pitävät enemmän tekstistä, joten voimme kääntää (translate, compile) ohjelmia konekielelle
- **Armlet on 16-bittinen arkkitehtuuri Scalalla simuloituna**
  - Käyttää edelliseissä harjoituksissa tehtyjä palikoita
  - Huom: Armlet on simuloitu laitteiston malli tietokoneen ohjelmistossa

Yleismaailmallisuus (Universality)

- **Ohjelmoitava kone, oikein ohjelmoituna, voi simuloida toista ohjelmoitavaa konetta**

Laskenta ja ohjelmointi ovat konsepteja, jotka ovat riippumattomia laitteesta ja ohjelmointikielestä

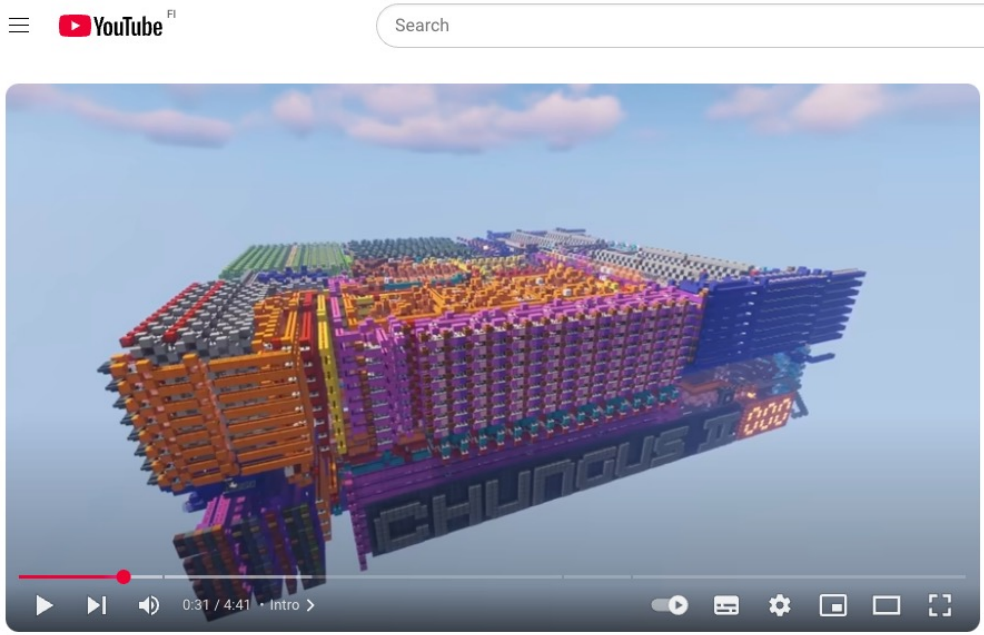
- **Pieni printti: fyysinen laite (esim muistin määrä) ja simulaation tehokkuus rajoittavat yllä olevaa hieman**

Yleismaailmallisuus (Universality)

- **Emulaattorit (emulators)**
- **Virtualisointi (virtualization)**
- **Kontit (containers)**
- **Mutta tietokoneet voi olla rakennettu myös muihin järjestelmiin, kunhan niillä on sekvenssilogiikkaa vastaava teho**

Yleismaailmallisuus (Universality)

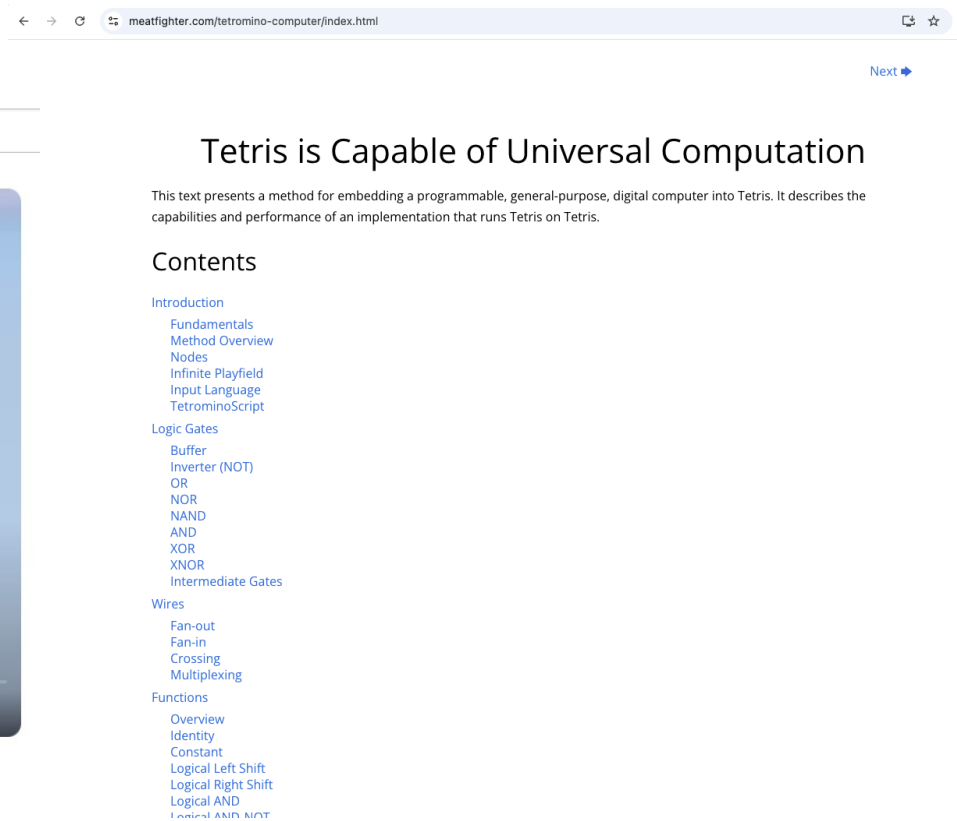
Esimerkiksi
Tietokone Minecraftin sisällä



CHUNGUS 2 - A very powerful 1Hz Minecraft CPU

https://youtu.be/FDiapbD0Xfg

tai perustuen tetrikseen?



https://meatfighter.com/tetromino-computer/index.html

Armlet ja sen Assembly-kieli

A”

Aalto-yliopisto
Perustieteiden
korkeakoulu

Mitä voimme tehdä `armletilla`?

- **Armletissa on**
  - Kahdeksan `var`:ia (rekisterit) ja yksi jättimäinen taulu (muisti)
  - ALU osaa aritmeettiset ja loogiset operaatiot
  - Ohjauslogiikka (control logic)
  - Operaatiot, jotka prosessori voi tehdä yhdessä kellosyklissä
- **Voimme antaa käskyjä datapolun kautta**
  - ALUlle: Boolen-operaatiot, yhteenlasku, vähentäminen…
  - Muistirajapintayksikölle: datan lataaminen ja tallentaminen
- **Käskyt suorituksen ohjaamiseen (execution control)**
  - arvojen vertailu ja haarautuminen

Kommentoi Assembly-koodisi!

- **Koodin kommentointi on aina tärkeää**
  - Muistathan O1:n tyylioppaan: https://plus.cs.aalto.fi/o1/2025/wNN/styleguide/
- **Tavallisessa koodissa on yleensä kahdenlaisia kommentteja:**
  - rajapinnan tarkoitus ja käyttötapa
  - metodien toteutus – selkeyttäviä asioita, ei jotain mitä on koodista suoraan luettavissa! Näitä pitäisi olla vain vähän

- **Assembly on sitten vähän eri asia…**

Kommentoi kaikki Assembly-koodisi!

- **Kommentoi JOKAINEN KOODIRIVI**
- **Omaksi parhaaksesi**
  - Assembly-kieli on hankalaa lukea, jopa minuutin päästä ihmiselle joka sen juuri kirjoitti
- **Muiden parhaaksi**
  - Koodin ihmislukijan on todella hankala seurata koodia
  - Assareiden mielenterveyden säilyttämiseksi – ennen kuin assari voi auttaa sinua harjoituksissa, koodisi on oltava kommentoitu. Muuten assari pyytää sinua kommentoimaan koodisi ensin (näin heidät on neuvottu tekemään)

Ensimmäinen Armlet-ohjelma

Scala

```
var t = 10
var i = 1
var s = 0

while t != 0 do
  s = s + i
  i = i + 1
  t = t - 1
// Now the sum is in 's'
```

Armlet

```
        mov $0, 10      # $0 = 10
        mov $1, 1       # $1 = 1
        mov $2, 0       # $2 = 0
@loop:                  # This is a label
        cmp $0, 0       # $0 == 0 ? (PS)
        beq >done       # if PS is eq
        add $2, $2, $1  # $2 = $2 + $1
        add $1, $1, 1   # $1 = $1 + 1
        sub $0, $0, 1   # $0 = $0 - 1
        jmp >loop       # Jump back
@done:                  # Another label
        hlt             # Stop, sum in $2
```

on kommenttimerkki

armlet-assemblyn perussyntaksi

- **Komennot ovat muotoa**
  - `kwd $L, $A, $B`
  - `kwd $L, $A, I`
  - `kwd $L, $A`
  - `kwd $A`
  - `kwd`
- **Kommenttimerkki #**
- **Esim.**
  - `Add $4, $0, $3 # laske yhteen $0 ja$3 ja`
    `# tallenna tulos $4:een`

- **Jossa**
  - `kwd` on kolmekirjaiminen lyhenne käskylle
  - `$L` on rekisteri, johon tulos tallennetaan (aina ensimmäisenä rekistereistä)
  - `$A` ja `$B` ovat syöterekisterit
  - `I` on välitön data (eli vakio)

Datapolun käskykanta (instruction set)

Rekistereissä oleville tiedoille

```
nop                 # no operation
mov $L, $A          # $L = $A (copy the value of $A to $L)
and $L, $A, $B      # $L = bitwise AND of $A and $B
ior $L, $A, $B      # $L = bitwise (inclusive) OR of $A and $B
eor $L, $A, $B      # $L = bitwise exclusive-OR of $A and $B
not $L, $A          # $L = bitwise NOT of $A
add $L, $A, $B      # $L = $A + $B
sub $L, $A, $B      # $L = $A - $B
neg $L, $A          # $L = -$A
lsl $L, $A, $B      # $L = $A shifted to the left by $B bits
lsr $L, $A, $B      # $L = $A shifted to the right by $B bits
asr $L, $A, $B      # $L = $A (arithmetically) shifted to the right by $B bits
```

vakioille

```
mov $L, I           # $L = I (copy the immediate data I to $L)
add $L, $A, I       # $L = $A + I
sub $L, $A, I       # $L = $A - I
and $L, $A, I       # $L = bitwise AND of $A and I
ior $L, $A, I       # $L = bitwise (inclusive) OR of $A and I
eor $L, $A, I       # $L = bitwise exclusive OR of $A and I
lsl $L, $A, I       # $L = $A shifted to the left by I bits
lsr $L, $A, I       # $L = $A shifted to the right by I bits
asr $L, $A, I       # $L = $A (arithmetically) shifted to the right by I bits
```

```
loa $L, $A          # $L = [contents of memory word at address $A]
sto $L, $A          # [contents of memory word at address $L] = $A
```


https://a1120.cs.aalto.fi/2026/notes/round-a-programmable-computer--processor.html#default-flow-of-execution

Elikkä, esimerkiksi

```
mov $0, 10        # $0 = 10
mov $1, 1         # $1 = 1
```

`mov` siirtää arvon rekisteriin, myös rekisteristä toiseen rekisteriin

```
add $1, $1, 1     # $1 = $1 +
sub $0, $0, 1     # $0 = $0 -
```

`add` laskee yhteen kaksi arvoa
`sub` on vähennyslasku

Loppululos tallennetaan rekisteriin, jonka numero on annettu ensimmäisenä

Elikkä, esimerkiksi

```
mov $0, 10      # $0 = 10
mov $1, 1       # $1 = 1
```

`mov` siirtää arvon rekisteriin, myös rekisteristä toiseen rekisteriin

```
add $1, $1, 1   # $1 = $1 +
sub $0, $0, 1   # $0 = $0 -
```

`add` laskee yhteen kaksi arvoa
`sub` on vähennyslasku

Loppululos tallennetaan rekisteriin, jonka numero on annettu ensimmäisenä

```
mov $1, 7
add $2, $1, 3
```

Mitä ovat rekisterien `$1` ja `$2` arvot näiden käskyjen jälkeen?

https://presemo.aalto.fi/o2fi2026

Hyppääminen (jumping) ja haarautuminen (branching)

- **Ohjelmat ovat dataa (kuten muistanet)**
  - Kun ohjelma koostetaan, se muutetaan binäärisanoiksi (dataksi)
  - Kun kone suorittaa ohjelmaa, se ladataan muistiin kuin mikä tahansa muu data
  - Ohjelmalaskuri (program counter, PC), on asetettu osoittamaan ensimmäistä käskyä, ja ohjelman suoritus alkaa tulkitsemalla tuossa kohtaa oleva data käskyksi
- **Muuttamalla PC:tä voimme muuttaa ohjeiden lukujärjestystä**
  - Hyppy- tai haarautumiskäskyssä asetetaan PC

```
00000: 0000000000011010
00001: 0000000000001010
00002: 0000000001011010
00003: 0000000000000001
00004: 0000000010011010
00005: 0000000000000000
00006: 0000000000100011
00007: 0000000000000000
00008: 0000000000100101
00009: 0000000000010001
00010: 0001010010000110
00011: 0000001001011110
00012: 0000000000000001
00013: 0000000000011111
00014: 0000000000000001
00015: 0000000000100100
00016: 0000000000000110
00017: 0000000000111111
```

Hyppy (jump) ja tunniste (label)

- **Hyppy siirtää argumentin osoitteen ohjelmalaskurin rekisteriin**

```
jmp >target # jump to label @target
# ... some code here ...
@target:
  # ... some further code here ...
```

- **Tämän vaikutuksesta ohjelman suoritus alkaa uudesta osoitteesta**
- **Yleensä ei tarvitse käsitellä absoluuttisia osoitteita, kääntäjä mahdollistaa nimikkeiden (label) käytön koodissa**
  - Nimike nappaa sen osoiteen, johon se on kirjoitettu
  - `@done`: Kääntäjä yhdistää tämän rivin muistiosoitteen tunnisteeseen ‘done’
  - Koodissa voi hypätä sekä eteen- että taaksepäin

JMP kysely

- **Miko on arvo rekisterissä $1 kun tämä koodi on suoritettu (alkaen ekasta rivistä)**

https://presemo.aalto.fi/o2fi2026

```
mov $1, 18      # $1 = 18
add $1, $1, 2   # $1 = $1 + 2
jmp >next       # Jump
mov $2, 10      # $2 = 10
sub $1, $1, $2  # $1 = $1 - $2
@next:
sub $1, $1, 5   # $1 = $1 - 5
hlt             # Stop
```

Mikä käsky suoritetaan seuraavaksi?

- **Oletetaan, että seuraava ohjelma on suoritettu alkusta alkaen kunnes se lopettaa**

```
mov $0, 3 # $0 = 3
jmp >there # Jump to label
@here:
add $0, $0, 1 # Increase by 1
hlt # Stop execution
@there:
sub $0, $0, 2 # Decrease by 2
jmp >here # Jump to label
hlt # Stop execution
```

- **Missä järjestyksessä laskukäskyt (add, sub) on suoritettu?**

https://presemo.aalto.fi/o2fi2026

Vertailu (comparison) ja haarautuminen (branching)

```
  cmp $7,0  # compare value in $7 to 0, this sets PS
  beq >done # jump to label @done if left == right, this sets PC
  # ... some code here ...
@done:
  # ... and more code here ...
```

- **Usein haluamme hypätä riippuen jonkin ehdon toteutumisesta**
- **Tässä prosessissa on kaksi vaihetta**
  1. Suoritetaan vertailu (comparison): rekisteri vs rekisteri/vakio
  2. Hypätään perustuen vertailun tulokseen (haarautuminen)
- **Hyppy tehdään, jos ja vain jos haarautumisen ehto on tosi**
- **Muuten suoritusta jatketaan oletusjärjestyksessä**

PS = prosessorin tila, processor status
PC = ohjelmalaskuri, program counter
cmp = compare
beq = branch if equal

Vertailu (comparison) ja haarautuminen (branching)

```
 # ... code
   # ... code
   # ... code
   # ... code
   cmp $0,0      # compare, this sets PS
   beq >done     # branch this sets PC
   # ... code - This will
   # ... code - only be
   # ... code - done
   # ... code - if $0 != 0
@done:
   # Start here if $0 == 0
```

- Melkein kuin:

```
if r7 != 0 then do <some> else do <other>
```

- Mieti: haarautuminen voi tehdä kumman vaan riippuen muusta koodista

```
        mov $0, 10       # $0 = 10
        mov $1, 1        # $1 = 1
        mov $2, 0        # $2 = 0
@loop:                   # This is a label
        cmp $0, 0        # $0 == 0 ? (PS)
        beq >done        # if PS is eq
        add $2, $2, $1   # $2 = $2 + $1
        add $1, $1, 1    # $1 = $1 + 1
        sub $0, $0, 1    # $0 = $0 - 1
        jmp >loop        # Jump back
@done:                   # Another label
        hlt              # Stop, sum in $2
```

cmp = compare
beq = branch if equal

Vertailu ja haarautuminen – sisäisesti

- **`cmp` päivittää prosessorin tila –rekisterin (processor status, PS, register)**
  - Kaikki eri vaihtoehdot huomioidaan
  - Tämän jälkeen rekistereiden arvoja voidaan muuttaa, mutta prosessorin tila ei muutu (ennen seuraavaa cmp-komentoa)
- **PS sisältää tuloksen uusimmasta `cmp`-käskyn kutsusta**
- **Haarautumiskäsky lukee PS:n ja hyppää sen mukaisesti**
  - Riippuen käskystä, katsotaan eri bitti
- **`cmp`-käskyn ja haaroittumiskäskyn välissä VOI olla koodia!**

Haarautumis-kysely

- **Mikä on rekisterin $0 arvo tämän koodin suorittamisen jälkeen (kun on aloitettu alusta)?** https://presemo.aalto.fi/o2fi2026

```
mov $0, 3        # $0 = 3
@start:
cmp $0, 2        # Compare val. in $0 to 2
sub $0, $0, 1    # $0 = $0 - 1
bne >start       # Branch if last cmp !=
hlt              # Stop
```

Käskyt suoritusjärjestykseen (execution flow)

`cmp $A, $B`	`# compare $A (left) and $B (right)`	Comparison
`cmp $A, I`	`# compare $A (left) and I (right)`	
`jmp $A`	`# jump to address $A`	Jump and branch based on the results of the latest comparison
`beq $A`	`# ... if left == right (in the most recent comparison)`	
`bne $A`	`# ... if left != right`	
`bgt $A`	`# ... if left > right (signed)`	
`blt $A`	`# ... if left < right (signed)`	
`bge $A`	`# ... if left >= right (signed)`	
`ble $A`	`# ... if left <= right (signed)`	
`bab $A`	`# ... if left > right (unsigned)`	
`bbw $A`	`# ... if left < right (unsigned)`	
`bae $A`	`# ... if left >= right (unsigned)`	
`bbe $A`	`# ... if left <= right (unsigned)`	
`jmp I`	`# jump to address I`	Jump and branch based on the results of the latest comparison
`beq I`	`# ... if left == right (in the most recent comparison)`	
`bne I`	`# ... if left != right`	
`bgt I`	`# ... if left > right (signed)`	
`blt I`	`# ... if left < right (signed)`	
`bge I`	`# ... if left >= right (signed)`	
`ble I`	`# ... if left <= right (signed)`	
`bab I`	`# ... if left > right (unsigned)`	
`bbw I`	`# ... if left < right (unsigned)`	
`bae I`	`# ... if left >= right (unsigned)`	
`bbe I`	`# ... if left <= right (unsigned)`	
`hlt`	`# halt execution`	Halt
`trp`	`# trap (break out of execution for debugging)`	

Kaikki käskyt

- **Kaikki armletin käskyt:**

  https://a1120.cs.aalto.fi/2026/notes/round-a-programmable-computer--armlet-programming.html#some-conventions-and-hints-for-the-exercises
  tai
  https://a1120.cs.aalto.fi/2026/notes/_downloads/06d2394ae9040ce7a57e67f0bd1378d7/armlet-quick-reference.pdf

- **Lisäksi ensin mainitulla sivulla on yleisiä tapoja ja vinkkejä**
  - Esim. mitä jos rekisterit ei riitä?

Datan lataus

Aalto-yliopisto
Perustieteiden
korkeakoulu

Datan lataaminen ja tallentaminen

- **Useimmat algoritmit tarvitsemat enemmän dataa kuin mitä mahtuu suorittimen rekistereihin**
- **Mitä data on?**
  - Bittejä, joita haluamme pystyä lataamaan rekistereihin ja joille haluamme suorittaa operaatioita
- **Tarvitsemme käskyt datan lataamisen muistista ja tallentamiseen**
  - `loa` ja `sto` –käskyjä voi käyttää siirtämään sanoja muistin ja rekisterien välillä
  - `loa x, y` # voisi vastata käskyä `x = mem(y)` ja
  - `sto x, y` # voisi vastatta käskyä `mem(y) = x`

Datan lataaminen ja tallentaminen

- **Muista, että muistia käytetään sekä ohjelmalle että datalle!**
  - Ero on vain miten tieto tulkitaan
  - Ei ole erillisiä alueita muistissa (ei ainakaan meidän arkkitehtuurissa)
  - Jos PC:ssä on tietty osoite, sieltä löytyvä sana tullaan tulkitsemaan käskynä!
- **`armletissa` PC alkaa aina 0:sta, joten datan pitäisi tulla ohjelman jälkeen**

Esimerkki: Arvojen lataaminen muistiin

- **Tämä ohjelma lataa arvon muistista, kasvattaa sitä 10:llä, ja kirjoittaa sen takaisin samaan muistipaikkaan:**

```
mov $0, >myDataStart   # Put address of data to $0
loa $1, $0             # $1 = value at address in $0
add $1, $1, 10         # Increase value in $1 by 10
sto $0, $1             # Store the value in $1 to address in $0
hlt                    # Stop
@myDataStart:
%data 5, 6
```

- **Tämän suorittamisen jälkeen arvo, joka on osoitteessa, johon `@myDataStart` on 15, ja sen jälkeisessä osoitteessa on 6**

Esimerkki 2: Arvojen lataaminen muistiin

- **Osoitteet ovat arvoja: osoitetta päivittämällä voi vaihtaa muistipaikkaa, josta ladataan/tallennetaan**

```
mov $0, >myDataStart   # Put address of data to $0
loa $1, $0             # $1 = value at address in $0
add $1, $1, 10         # Increase value in $1 by 10
add $0, $0, 1          # Update the address in $0 by 1. <- MUUUTOS!
sto $0, $1             # Store the value in $1 to address in $0
hlt                    # Stop
@myDataStart:
%data 5, 6
```

- **Tämän suorittamisen jälkeen @myDataStart on 5 ja seuraava 15**
- **Käsky (ja vain se) määrittelee, miten arvo tulkitaan**
  - `sto $1, $0`, olisimme kirjoittaneen $1:n arvon osoitteeseen 15

loa -kysely

- **Mitkä on arvot rekistereissä $1, $2, ja $3 tämän koodin suorittamisen jälkeen?** https://presemo.aalto.fi/o2fi2026

```
mov $0, >myDataStart   # Put address of data to $0
loa $1, $0             # $1 = value at address in $0
add $0, $0, 1          # Increase address
loa $2, $0             # $2 = value at address in $0
add $0, $0, 2          # Increase address
loa $3, $0             # $3 = value at address in $0
hlt
@myDataStart:
%data 5, 6, 7, 8, 9, 10
```

Datan lataaminen ja tallentaminen

- **Direktiivit (directive) ovat assemblyn aputyökaluja datan lisäämiseksi muistiin**
  - Alkavat %-merkillä
  - Seuraavalla dialla `%data 20` tarkoittaa, että muuta literaali 20 binääriksi ja kirjoita se dataksi muistiin, siinä vaiheessa kun assembly-ohjelma on käännetty konekoodiksi

- **Kurssimateriaalista löytyy lisää esimerkkejä**

Esim: datan lataaminen muistista

```
# Computes (to $0) the sum of the data and then halts.

# Let us first set things up ...
      mov $0, 0                    # the sum starts with 0
      mov $2, >length_of_my_data   # set up memory address where to get length
      loa $2, $2                   # load the length from memory to $2
      mov $1, >my_data             # set up memory address where to get the data

# Now let us loop through the data and accumulate the sum ...
@sum_loop:
      cmp $2, 0                    # compare length with 0
      beq >done                    # ... branch to label @done if $2 == 0
      loa $3, $1                   # load a data item from memory
      add $0, $0, $3               # accumulate the sum in $0
      add $1, $1, 1                # advance to next data item
      sub $2, $2, 1                # decrement length by one
      jmp >sum_loop                # continue the summation

# ... until we are done
@done:
      hlt                          # the processor stops here

# Our data follows the program code in the binary ...
@length_of_my_data:
      %data 20
@my_data:
      %data 296, 573, 291, 415, 825, 674, 426, 632, 793, 701, 884, 1, 989, 912, 254, 869, 462, 296, 767, 220
```

Aluksi `$2` on muistiosoite,
Sitten katsotaan mikä arvo muistiosoitteessa olikaan ja laitetaan se rekisteriin `$2`

`$1` sisältää muistiosoitteen datan ekalle alkiolle

Kunnes kaikki alkiot on käyty
- lataa uusi alkio muistista
- tee yhteenlasku
- päivitä muistiosoite
- päivitä laskuri

Osaatkos lukea koodia? Muistathan kommentit

https://presemo.aalto.fi/o2fi2026

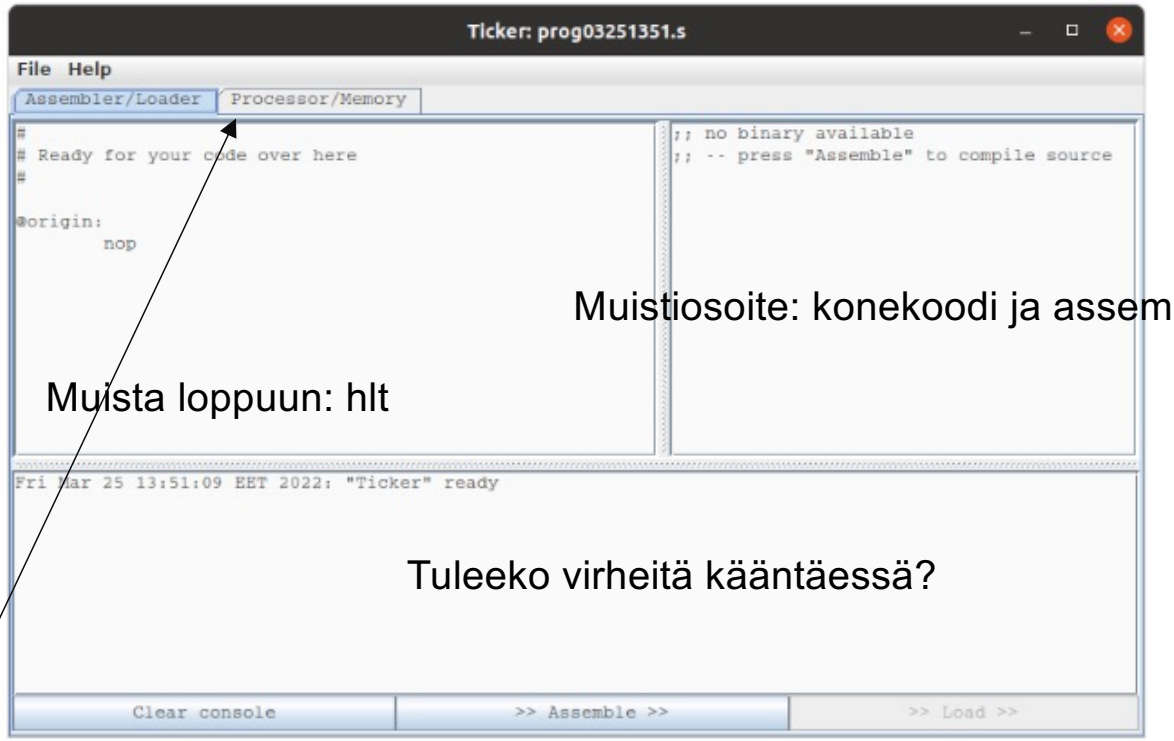
- **Mitä seuraava ohjelma tekee?**
- **Mitä on talletettu rekistereihin $0, $1 ja $2, kun suoritus pysähtyy?**
- **Avuksi**

`blt X`	Hyppää X:ään jos vasen < oikea (edellä `cmp`)
`cmp $A, $B`	Vertaile rekistereitä A ja B
`hlt`	Lopeta suoritus
`jmp X`	Hyppää (eli mene) X:ään
`lsl $L, $A, $B`	L = A << B (vasen siirto)
`mov $L, I`	L = joku_välitön_data

```
 mov $0, 18
 mov $1, 15
 cmp $0, $1
 blt >labA
 mov $2, $0
 jmp >labB
@labA:
 mov $2, $1
@labB:
 lsl $2, $2, 1
 hlt
```

Kokeille Ticker-ohjelmalla

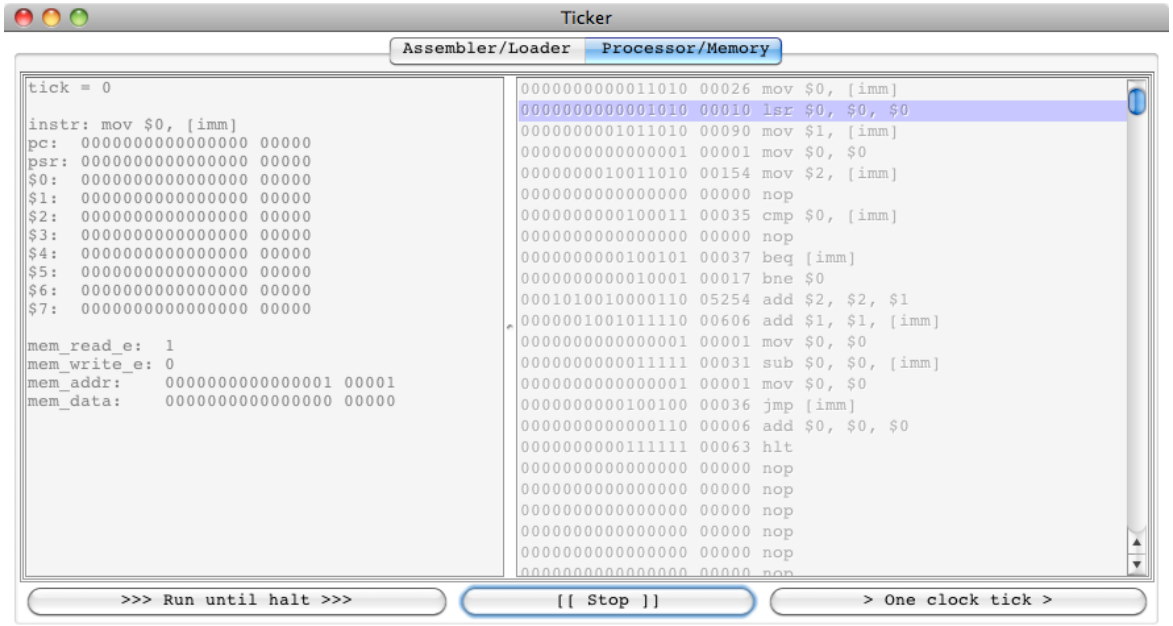
- **Harjoitusmateriaalista löytyy `ohjelma launchTicker.scala` armlet-aliprojektista**
- **Konsolissa:** import armlet.*
  new Ticker()
  - Ensin koodi vasemmalle
- **Assemble**
  - tuliko virheilmoituksia?
- **Load**
- **Sitten vaihda tab ylhäältä!**



https://aalto.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=c4b5f41a-2728-4231-bdcf-b274007762e5

Kokeile Ticker-ohjelmalla

- **Ajo-moodi (Processor/Memory-tab)**
  - Vasemmalla prosessori ja kaikki sen rekisterit
  - Oikealla muisti (binääri, heksa, desimaali, ja lopuksi miten tulkitaan käsky)
- **Ajaminen kokonaan tai kellosykli kerrallaan**



https://aalto.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=c4b5f41a-2728-4231-bdcf-b274007762e5

Voimmeko ilmaista (express) enemmän Scalalla kuin Assembly-kielellä?

- **Periaatteessa EI**
  - Kaikki “korkean tason” ohjelmat käännetään konekoodiksi
- **Käytännössä KYLLÄ**
  - Abstraktiot, tietotyypit, ohjelmoinnin rakenteet, kirjastot…
  - Tuottavuus on suurempi korkean tason kielillä

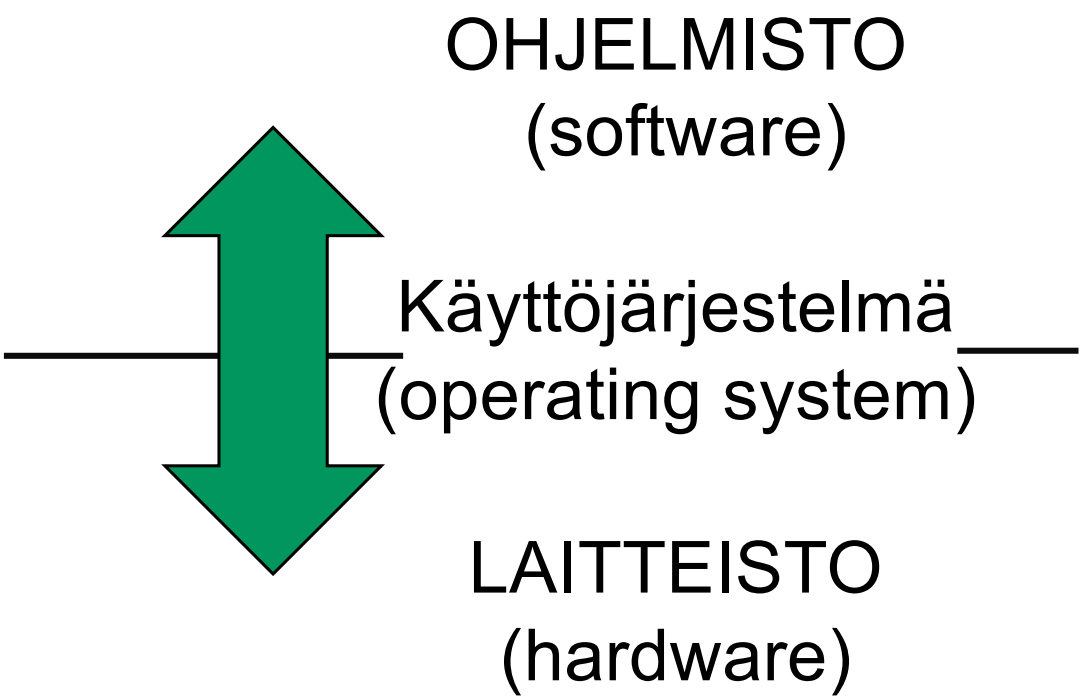
Miksi opiskella Assemblyä?

- **Lähempänä ‘rautaa’**
  - Laitteistoa ohjataan suoraan
  - Tehokasta esim. muistia säästävää
- **Päästää laitteiston tehot valloilleen**
- **Koneiden “äidinkieli”**
- **Mutta se on ohjelmoijalle aika hankalaa**

- **MUISTA KOMMENTOIDA KOODISI! ☺**

Mitä olemme jättäneet pois…

- **Kuinka ohjelmat toimivat yhtäaikaisesti (simultaneously)?**
- **Miten ohjelma voi odottaa syötettä käyttäjältä?**
- **Miten ohjelma voi tulostaa?**
- **Miten korkean tason ohjelmat käännetään konekielelle?**
- **Miten erottaa eri ohjelmien data niin, että ne eivät näe eri ohjelman dataa?**

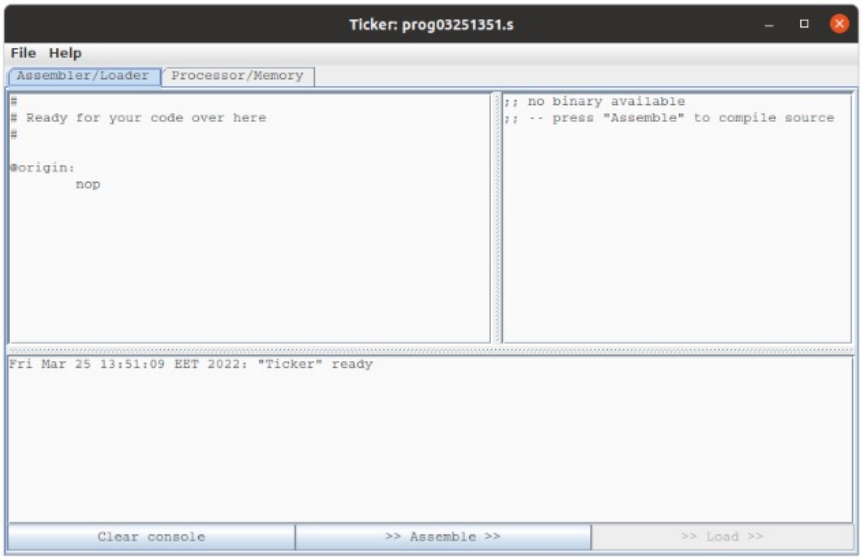


Kurssin osan 1 yhteenveto

- Lopulta, kaikki on bittejä
- Kombinaatiologiikkaa voidaan käyttää muuttamaan bittejä
- Sekvenssilogiikka mahdollistaa tilan ylläpitämisen
- Tietokoneen arkkitehtuuri (computer achitecture) mahdollistaa koneen suunnittelun
- Ohjelmoitava kone suorittaa ohjelmia automaattisesti
- Ohjelma voi muuttaa kontrollivuotaan (control flow) haaroittumalla tilaan perustuen
- Ohjelmointikielet (kuten Scala) ovat “vain” ohjelmistotyökaluja, jotka auttavat valjastamaan laitteiston tekemään mitä ohjelmoija haluaa

Harjoituksissa tällä kierroksella

- **Käytä `Armlet Tickeriä`**



https://a1120.cs.aalto.fi/2026/notes//round-a-programmable-computer--armlet-programming.html#a-playful-introduction-to-ticker

Harjoituksissa tällä kierroksella

1. Lausekkeen evaluointi
2. Sana-operaatioita
3. Arvojen alue data-taulukossa
4. Yleisin arvo data-taulukossa
5. Kertolasku
6. Jakojäännös
7. Haastetehtävä: 32-bittinen jakojäännös
8. Haastetehtävä: keskiarvo
9. Haastetehtävä: suurin yhteinen jakaja

Harjoituksissa tällä kierroksella

- **Kommentoi koodisi – omaksi parhaaksesi!**
  - Assistantit eivät voi auttaa, ennen kuin olet kommentoinut koodisi!
- **Käy läpi ensin kurssimateriaalin esimerkit!**
  - Sisältävät hyödyllistä koodia
- **Meillä on vain kahdeksan rekisteriä (`$0`-`$7`) – huomaa ero:**
  - Rekisteri `$8` aiheuttaa Tickerissä virheen (`failure: string matching…`)
- **Algoritmin suunnittelu ensin (ja ehkä jopa Scala-ohjelman teko)**
  - Osassa saa käyttää vain tietyn määrän kellosyklejä
- **Käytä Tickeriä ohjelmasi debuggaamiseen**
- **Tällä kierroksella ei ole yksikkötestejä!**
  - Mutta harjoituksissa on kommenteissa esimerkkejä