

Computersystemen en -architectuur

MIPS Game of Life

1 Ba INF
2025–2026

Tim Apers

Assistent

tim.apers@uantwerpen.be

Tibo Verreycken

Student-assistent

tibo.verreycken@student.uantwerpen.be

Vrijdag 5 december 2025

1 Introductie

Deze opdracht is gebaseerd op het structureel kader van het vorige project (doolhof / videospel) maar vervangt de concrete game door een implementatie van John Conway's *Game of Life*. Je gaat werken in MIPS-assembler en je code wordt geëvalueerd in MARS. Gebruik de templatefile `start.asm` op MSDL als beginpunt. Voeg duidelijke comments toe, structureer je code in functies en maak voor elke functie correcte stackframes aan.

2 Algemene aanwijzingen

- Lever je eind oplossing in via Inginious: “CSA” → “MIPS” → “MIPS Game of Life” vóór **zondag 21 december 2025, 22:00**.
- Tijdens de beoordeling wordt je programma geevalueerd met MARS met de ingewerkte Bitmap Display. Zorg er dus voor dat je project correct werkt in MARS.
- Gebruik registers volgens de conventies: argumentregisters `$a0–$a3`, return in `$v0/$v1`, temporary registers `$t0–$t9`, etc.

Onderdeel	Score
Correct inlezen en weergeven van start state	4
Dynamisch inlezen van meerdere start states	6
Correcte implementatie van de Game of Life regels	6
Juiste timing van de regels (geen in-place corruptie)	6
Periodieke grenscondities (wrap-around)	5
Implementatie van command-line argument “r” (reverse)	5
Correct gebruik van stackframes	4
Code kwaliteit en comments	4
Totaal	40

Tabel 1: Puntentabel voor de Game of Life opdracht

3 Opdracht

Je ontwikkelt een MIPS-programma dat John Conway's *Game of Life* simuleert en visualiseert. Het programma maakt gebruik van het bitmap-geheugen zoals aangehaald in de slides van de praktijksessies. De opdracht is opgedeeld in de volgende onderdelen.

3.1 Inlezen van de start state (10%)

Het eerste onderdeel van de opdracht bestaat uit het inlezen van een startbestand dat de initiële toestand van de cellen voorstelt. Specifiek schrijf je een oplossing die een tekstbestand, **input_1.txt** in dit geval, opent en regel voor regel uitleest.

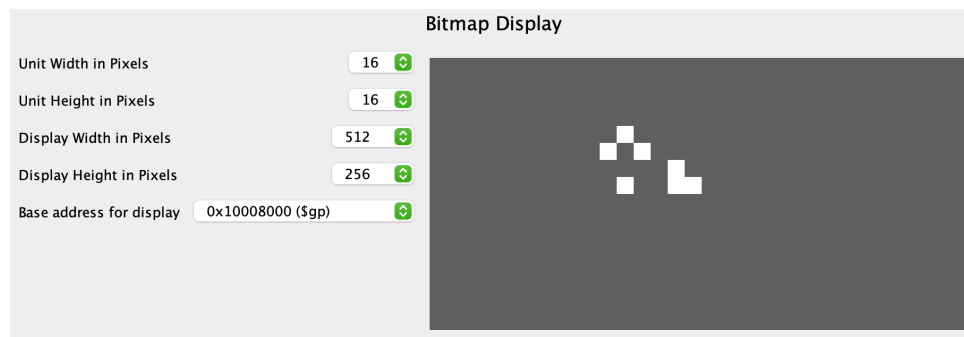
Het bestand bevat uitsluitend ASCII-tekens '0' en '1', evenals newline-tekens. Elk karakter in het bestand correspondeert met één cel in het raster. Een '0' stelt een dode cel voor, terwijl een '1' een levende cel voorstelt. Het rasterformaat is identiek aan dat van de vorige opdracht, met hetzelfde aantal rijen en kolommen.

Character	Betekenis	Kleur
0	dood	zwart
1	levend	wit
\n	einde van de celrij	-

Om de rij- en kolomnummers om te zetten naar de juiste geheugenlocaties in het bitmap-geheugen, kan je gebruik maken van de functies uit deel 1. De bovenste rij van het raster heeft rijnummer 0 en de meest linkse kolom heeft kolomnummer 0. Voor een raster van 4 bij 5 cellen ziet dat er als volgt uit:

(0, 0)	(0, 1)	(0, 2)	(0, 3)
(1, 0)	(1, 1)	(1, 2)	(1, 3)
(2, 0)	(2, 1)	(2, 2)	(2, 3)
(3, 0)	(3, 1)	(3, 2)	(3, 3)
(4, 0)	(4, 1)	(4, 2)	(4, 3)

Zoals reeds vermeld in de vorige opdracht, zal de pixel die overeenkomt met rij 0 en kolom 0 dus opgeslagen zijn op het geheugenadres in register **\$gp**. Na een correcte implementatie, heb je volgende voorstelling op je display:



Figuur 1: Bitmap Display na het inladen van input_1.txt

3.2 Dynamisch inlezen van meerdere start-states (15%)

In dit onderdeel breid je je programma uit zodat het meerdere startbestanden kan inlezen, zoals `input_1.txt`, `input_2.txt` en `input_3.txt`. Zorg ervoor dat alle cellen correct worden weergegeven op het display en dat het programma flexibel meerdere bestanden kan verwerken.

3.3 Implementatie van John Conway's regels (30%)

In dit onderdeel ga je de klassieke regels van John Conway's *Game of Life* implementeren. Het doel is dat het programma voor elke cel bepaalt of deze levend of dood blijft in de volgende generatie. De regels zijn als volgt:

1. Een levende cel met 2 of 3 levende burens blijft leven.
2. Een levende cel met maar 0 of 1 levende burens sterft door vereenzaming; met 4 of meer burens sterft ze door overbevolking.
3. Een dode cel met precies 3 levende burens wordt levend.

Om correcte resultaten te garanderen, mag je de states van de cellen **niet** in-place updaten tijdens het scannen van het bord. Dit zou leiden tot verkeerde berekeningen, omdat burens mogelijk al zijn aangepast. Ontwerp daarom een methode die deze veranderingen opslaat en deze toepast nadat het volledige bord is gescand.

Voor dit onderdeel gebruik je de voorbeeldbestanden, `input_1.txt`, `input_2.txt`, en het voorziene **update** label. Test je functie door te controleren of het raster correct wordt bijgewerkt volgens de regels van de Game of Life en dat de veranderingen pas zichtbaar zijn nadat het volledige bord is verwerkt.

Na een correcte implementatie van de regels verdwijnen alle levende cellen na precies 12 stappen bij `input_1.txt`, terwijl de configuratie uit `input_2.txt` onbeperkt behouden blijft.

3.4 Periodische grenscondities (wrap-around) (12,5%)

In dit onderdeel breid je je implementatie verder uit door correcte randafhandeling te voorzien. Het standaardgedrag waarbij cellen aan de randen minder burens hebben, leidt tot ongewenste en niet-representatieve simulaties. Daarom pas je periodische grenscondities toe: het raster wordt behandeld als een tweedimensionale torus waarbij de bovenkant aansluit op de onderkant en de linkerzijde op de rechterzijde. Hierdoor heeft elke cel steeds exact acht burens.

Technisch gezien betekent dit dat je bij het bepalen van de buurcellen telkens moet “wrappen” wanneer een index buiten het raster valt. Dit kan gerealiseerd worden door te werken met geheugenoffsets ten opzichte van `$gp` en vervolgens een modulo-operatie toe te passen met de totale rasterlengte L (het aantal cellen in het raster):

$$\text{offset} = (\text{adres} - \$gp) \bmod L$$

$$\text{effectief adres} = \$gp + \text{offset}$$

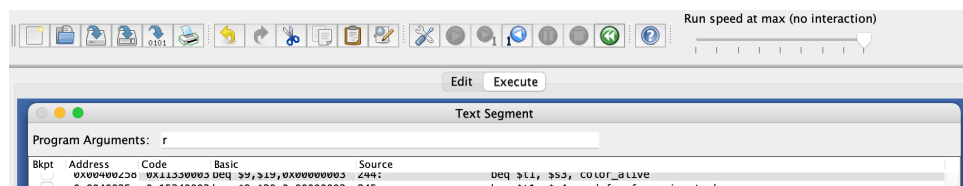
Het adres is hierbij de geheugenlocatie van de zagezegde buur van de cell ten opzichte van \$gp. Door deze aanpak worden alle geheugenadressen die buiten het raster zouden vallen automatisch correct teruggebracht binnen de geldige grenzen.

Na een correcte implementatie van de grenzen zal de figuur van `input_3.txt` blijven ronddwalen in het raster.

3.5 Ondersteuning voor het command-line argument **r** (reverse) (12,5%)

In dit onderdeel voorzie je je programma van ondersteuning voor een optioneel command-line argument. Wanneer het programma wordt opgestart met het argument `r`, moeten de kleuren van levende en dode pixels omgedraaid worden: levende cellen worden weergegeven in het grijs en dode cellen worden nu wit.

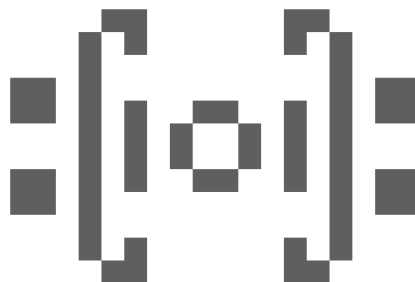
In de MARS-omgeving kan je dit soort argumenten toelaten door het volgende aan te vinken: *Settings* → *Program arguments provided to MIPS program*. Na het assembleren verschijnt er bovenaan in het Text segment een tekstveld met de titel *Program Arguments:*, waarin je eenvoudig de letter **r** kan invullen.



Figuur 2: MARS dashboard als program arguments toegelaten zijn.

Voor deze opdracht hoef je slechts één karakter te controleren. Wanneer het programma wordt gestart met command-line argumenten, plaatst MARS het aantal argumenten in register **\$a0**. Dit is dus het aantal meegegeven woorden (zoals **r**, **test**, ...). Het adres van het eerste argument wordt in **\$a1** geplaatst; dit is een pointer naar een reeks null-terminated strings waarin alle argumenten na elkaar staan opgeslagen.

Je leest dus enkel het eerste teken van het eerste argument uit. Vergelijk dit karakter met de ASCII-waarde van de letter 'r'. Indien beide overeenkomen, activeer je de reverse-modus. In alle andere gevallen (geen argument aanwezig of een ander teken dan **r**) gebruik je de standaard-kleurweergave. **input_2.txt** wordt dan:



3.6 Stackframes en codeorganisatie (10% en 10%)

Een belangrijk deel van deze opdracht bestaat uit het schrijven van gestructureerde, duidelijke en correct gedocumenteerde MIPS-code. In dit onderdeel wordt vooral beoordeeld hoe je functies opbouwt, hoe je met de stack omgaat en hoe overzichtelijk je programma georganiseerd is.

- **Correct gebruik van stackframes:** Maak duidelijk onderscheid tussen functies die een volledig stackframe vereisen en functies waarvoor dit niet strikt noodzakelijk is. Alle functies die argumenten verwerken, registers overschrijven, lokale variabelen gebruiken of andere functies aanroepen, moeten een correcte prolog en epilog implementeren. In de prolog bewaar je de callee saved registers die je functie nodig heeft (i.e. `$s0–$s7`), de framepointer `$fp`, en indien de functie andere functies oproept, ook `$ra`.

In de epilog herstel je de registers in omgekeerde volgorde, herstel je de oorspronkelijke waarde van de stackpointer, en keer je terug via `jr $ra`. Kleinere hulpfuncties die geen registers overschrijven (buiten `$t`-registers) en geen subroutines aanroepen, mogen eventueel zonder stackframe geschreven worden, maar alle belangrijke, herbruikbare functies moeten de conventies uit de theorie- en practicumlessen strikt volgen.

- **Heldere codeorganisatie:** Deel je programma op in duidelijke, herbruikbare functies. Typische voorbeelden zijn: het berekenen van geheugenadressen, het tellen van levende burens, het inlezen van een tekstbestand, of het updaten van het bitmap-geheugen.
- **Goede documentatie en commentaar:** Schrijf bondige maar duidelijke commentaarregels boven functies en bij complexe instructiereeksen. Leg uit *wat* de functie doet en waarom bepaalde stappen nodig zijn. Vermijd overbodige commentaar, maar zorg ervoor dat de structuur van het programma meteen begrijpelijk is voor een lezer.
- **Netheid en consistentie:** Gebruik een consistente indentatie, zorg ervoor dat labels logisch benoemd zijn en dat je functies enkel die verantwoordelijkheden hebben die hun naam suggereren. Een overzichtelijke opbouw maakt debugging eenvoudiger en verhoogt de kwaliteit van je oplossing.

Veel succes! Als je vragen hebt over details van de fileformaten of de gebruikte registers, kijk terug naar de slides en de template op MSDL of contacteer de assitent!