

Inhaltsverzeichnis

1 Einleitung	3
1.1 Vorgehen und Arten von Tests	3
2 Funktionale Tests	4
2.1 Komponententests	4
2.1.1 Controller - Simulation	4
2.1.2 View	6
2.1.3 JUnit-Überdeckung	6
2.2 Integrationstests	14
2.2.1 Große JUnit Tests	14
2.3 Systemtests mit manuellen Testprotokollen	15
2.3.1 MIMA bauen:	16
2.3.2 MIMA Programm schreiben:	19
2.3.3 MIMA-Programm im Step-Modus ausführen	21
2.3.4 Goggle-Test	24
2.3.5 RISC-V-Computer bauen	26
2.3.6 RISC-V-Beispielprogramm laden und compilieren	27
2.3.7 RISC-V-Programm im Fast-Modus ausführen	27
3 Nicht-funktionale Tests	30
3.1 User Tests	30
3.1.1 Einige Stimmen unserer Tester	30
3.1.2 Feature Requests der Tester	31
3.1.3 Von Testern gefundene Bugs	31
3.2 Leistungstests	31
3.2.1 MIMA-Leistungstest	32
3.2.2 RISC-V-Leistungstest	32
3.2.3 Speicher-Lasttest	33
4 Sonstiges	34
4.1 Verbliebene Bugs	34
4.2 Statische Tests	34
4.3 Testautomatisierung	34
5 Glossar	35

1 Einleitung

Zum Ende des Projekts soll die Modifikation getestet werden. So kann zwar nicht die Abwesenheit von Fehlern verifiziert werden, allerdings kann die Zahl der Fehler drastisch reduziert werden, sodass die Stabilität des Projekts erhöht werden.

1.1 Vorgehen und Arten von Tests

Bereits während der Implementierung wurden ersten Komponententests insbesondere von wichtigen Methoden erstellt. Im Rahmen der Validierung wurde dies um alle öffentlichen Methoden ergänzt, welche nicht im View befindlich sind. Im View konnte nur ein Teil der Methoden abgedeckt werden. Als Ergänzung zu Komponententests wurden abgewandelte Integrationstests erstellt. Schließlich wurden auch manuelle Systemtests entworfen und durchgeführt. Ergänzt wurden diese funktionalen Tests durch Nutzer-Tests und Leistungstests.

2 Funktionale Tests

Die funktionalen Tests für das Projekt lassen sich in drei Kategorien teilen. Zunächst wurden noch während der Entwicklung Komponententests für wichtige Methoden und Klassen geschrieben und später um Tests für alle weiteren öffentlichen Methoden ergänzt. Anschließend wurde mithilfe von JUnit5 umgesetzt.

2.1 Komponententests

2.1.1 Controller - Simulation

Risc-V-Tests

In den Tests wurde die korrekte Ausführung einzelner Logik Befehle getestet. Dies wurde mit Blackbox-Tests erreicht, da eine whitebox Abdeckung aufgrund der Implikation des Instruction-Set-JSON nicht sinnvoll gewesen wäre. Um alle relevanten Fälle abzudecken, wurde ein Python Script genutzt, um JUnit Tests zu schreiben. Die erwarteten Werte erhalten wir durch Ausführen des Assembler Codes in RARS, einem Open-Source RISC-V Simulator. Die getesteten Instruktionen wurden jeweils mit 0, 1, -1 und 2047 (1000 bei 12 Bit, da diese als einzige Größe Schwierigkeiten erzeugt) in allen Attributen getestet. Manche Werte sind bei einigen Instruktionen natürlich nicht möglich, z.B. kann man bei *slli t0, t1, t2* nicht um mehr als 32 Bit oder weniger als 0 Bit shiften. Bei solchen Instruktionen wurden die nicht möglichen Werte nicht mit einbezogen. Bei Instruktionen mit Registern als Ursprungswert wurde vorher der zu testende Wert mit *addi* in ein Register gespeichert, welches dann als Argument für die eigentliche Instruktion genutzt wurde. Bei der Ausführung der einzelnen Testfälle konnte leider nicht der SimulationTimeHandler genutzt werden, da dieser die Simulation in einem anderen Thread ausführt, was mit bei JUnit zu unendlicher Ausführung geführt hat. Deshalb wurde nur der SimulationSequenceHandler in diesen Tests mitgetestet.

Executor-Tests

In den Tests des Executors wurde geprüft, ob die verschiedenen Typen von Mikroinstruktionen korrekt ausgeführt werden. Dabei wurden die folgenden Fälle geprüft:

- Alu-Instruktion ohne Speicher-Instruktion und mit korrekten Eingaben
- Alu-Instruktion mit Speicher-Instruktion und mit korrekten Eingaben

- Alu-Instruktion PAUSE
- Datenbewegungs-Instruktion mit korrekten Eingaben
- Bedingte Instruktion mit Vergleich mit Vorzeichen und wahrer Bedingung
- Bedingte Instruktion mit Vergleich mit Vorzeichen und falscher Bedingung
- Bedingte Instruktion mit Vergleich ohne Vorzeichen und wahrer Bedingung
- Bedingte Instruktion mit Vergleich ohne Vorzeichen und falscher Bedingung

Die verschiedenen Aktionen, die in Alu-Instruktionen möglich sind, wurde nochmals gesondert im AluController getestet. Dafür wurden 40 Tests von Hand konstruiert.

Clustering-Tests

In den Tests zum Clustering wurde geprüft, ob sich die Blöcke korrekt zu Clustern verbinden und dass sich Cluster korrekt bei der Zerstörung eines Blocks auf trennen. Dabei wurden die folgenden Fälle geprüft:

- Bus-Blöcke verbinden sich miteinander und mit Nicht-Bus-Blöcken des selben Clusters
- Nicht-Bus-Blöcke verbinden sich nur mit genau einem Bussystem
- Nicht-Bus-Blöcke können mehrfach in einem Bussystem verbunden werden
- Bus-Blöcke verbinden sich nicht mit Nicht-Bus-Blöcken eines anderen Bussystems
- Zwei Bussysteme werden bei dem verbinden durch einen Bus-Block korrekt zusammengeführt
- Beim Verbinden von zwei Bussystemen mit einem Bus-Block werden auch notwendige Verbindungen zwischen Bus-Blöcken und Nicht-Bus-Blöcken ergänzt
- Blöcke können durch Zerstörung aus dem Bussystem entfernt werden
- Das Bussystem trennt sich bei Zerstörung der Verbindung zweier Bussysteme auf
- Nach dem Auftrennen eines Bussystems sind alle Nicht-Bus-Blöcke nur einem Bussystem zugehörig ggf. durch Entfernen von Verbindungen
- Wenn das Bussystem eines Nicht-Bus-Blocks zerstört wird, versucht der Nicht-Bus-Block sich mit einem anderen anliegenden Bussystem zu verbinden

Bei den Tests zum Clustering wurde zeitgleich auch die Graph-Repräsentation im BusSystem-Model mitgetestet, da diese die Daten für das Clustering hält.

2.1.2 View

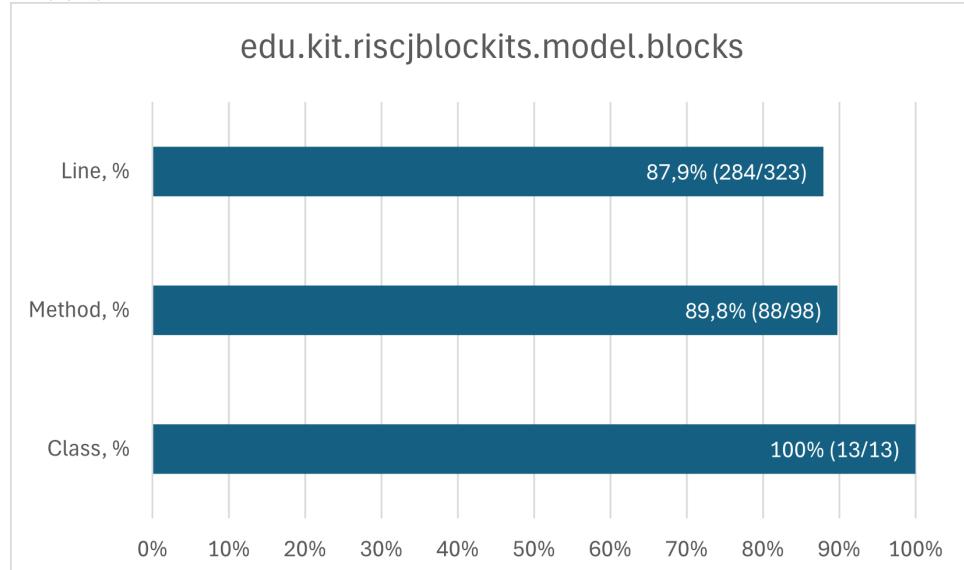
Komponententests im View erwiesen sich als schwierig, da alle Klassen viele Abhängigkeiten aus Minecraft besitzen. Um viele Methoden zu testen, brächte man eine laufende Minecraft Instanz “im Hintergrund“. Das ist nicht möglich. Allerdings gibt es eine Möglichkeit zwar nicht eine Welt im Hintergrund laufen zu haben, aber einige wichtige Bestandteile wie die Block Registry initialisiert zu haben. Fehlende Klassen, wie `minecraft.world`, werden dann mit Mockito erstellt. So kann Block und Funktionalität, die nicht direkt mit Minecraft interagiert, getestet werden. Zum Beispiel wird getestet: Dass Block Controller richtig erzeugt werden. Das die Brille den richtigen Text bekommt. Das ein Block mit Inventar Items richtig verwaltet.

2.1.3 JUnit-Überdeckung

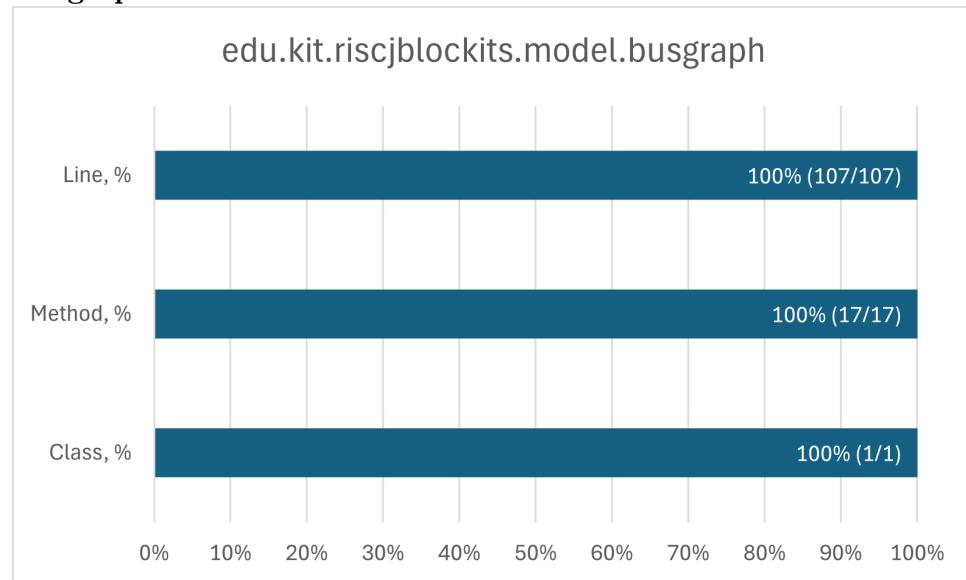
Im Folgenden wird die Überdeckung durch JUnit-Tests nach Paketen aufgeteilt vorgestellt.

Model

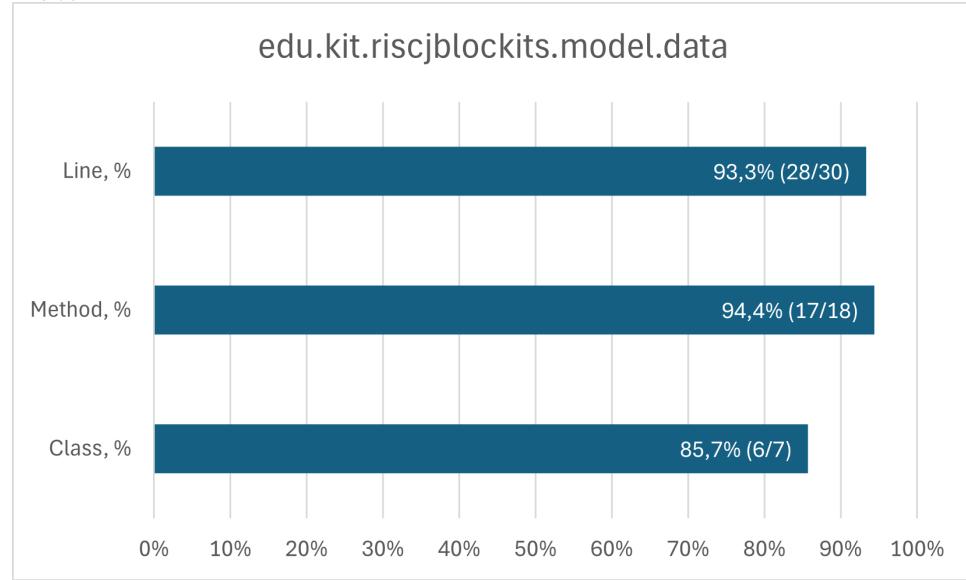
Blocks



Busgraph

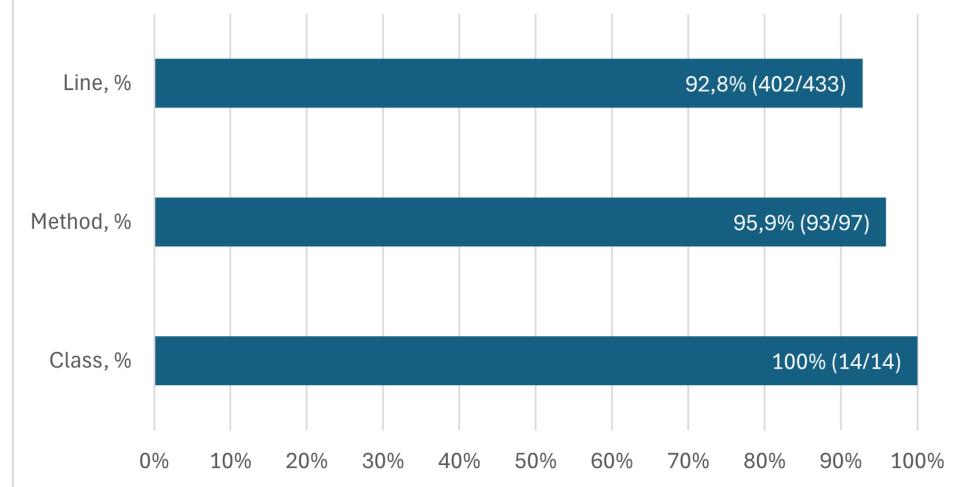


Data



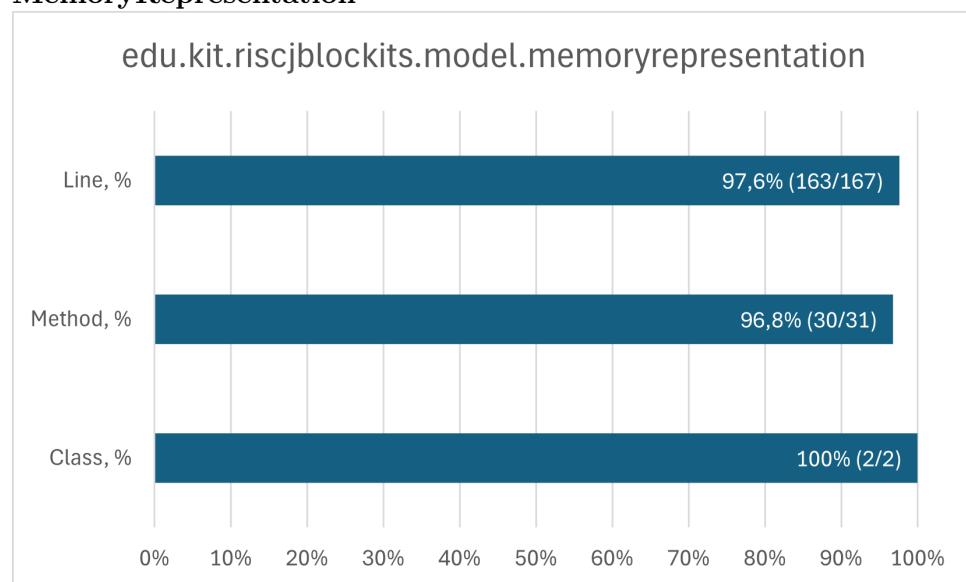
InstructionSet

edu.kit.riscjblockits.model.instructionset



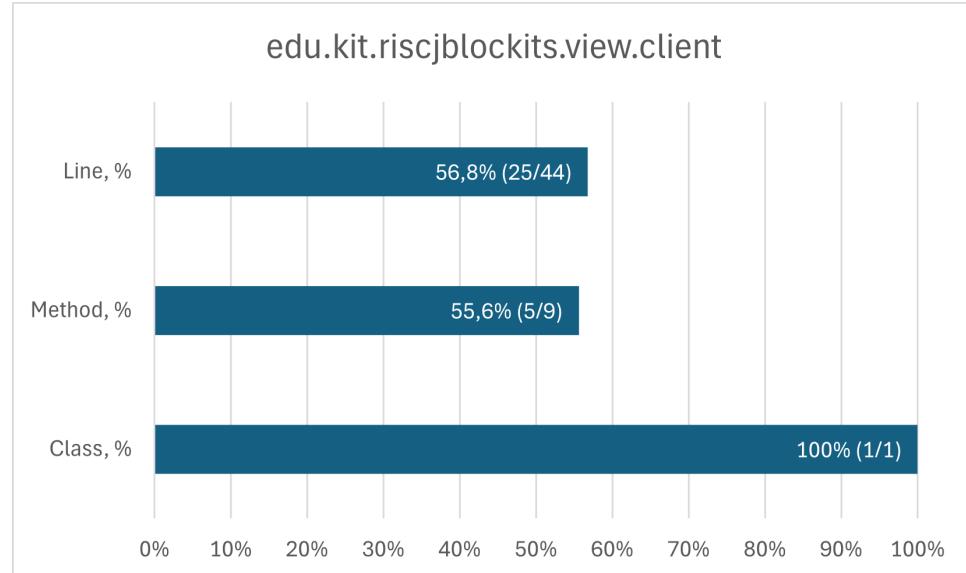
MemoryRepresentation

edu.kit.riscjblockits.model.memoryrepresentation

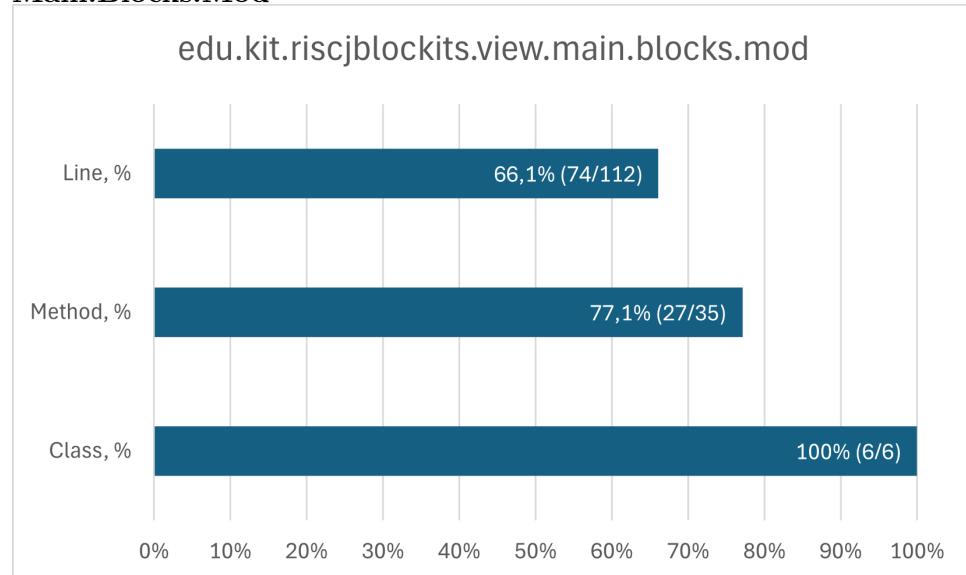


View

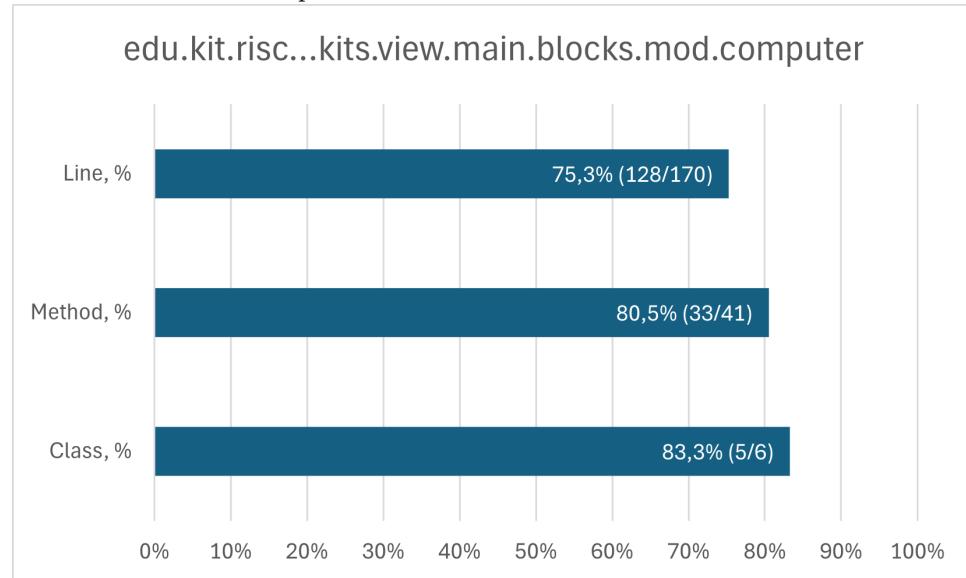
Client



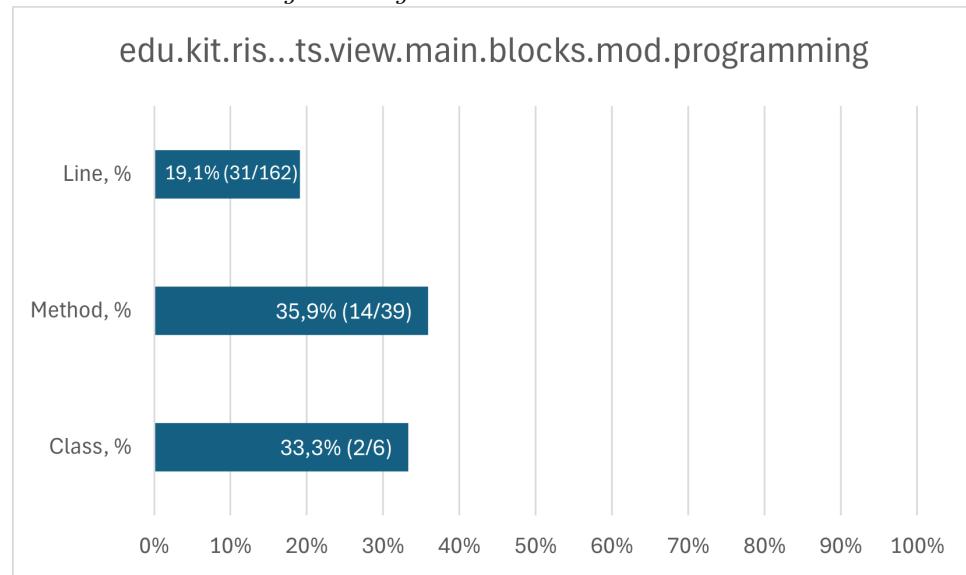
Main.Blocks.Mod



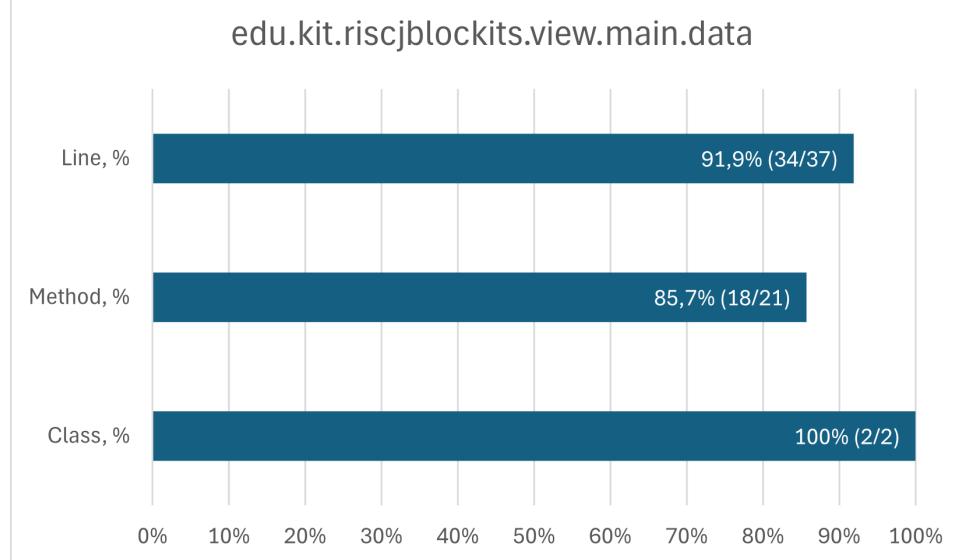
Main.Blocks.Mod.Computer



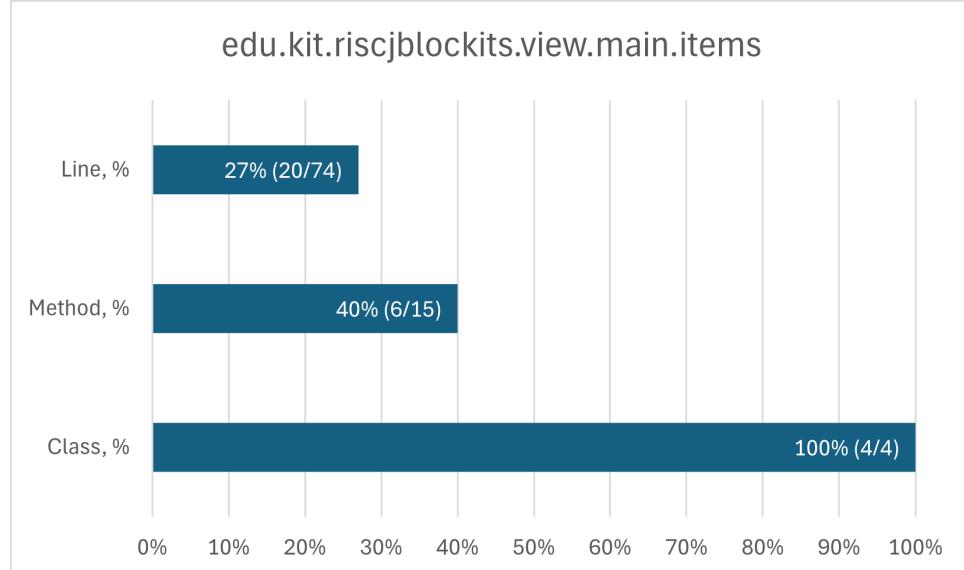
Main.Blocks.Mod.Programming



Main.Data

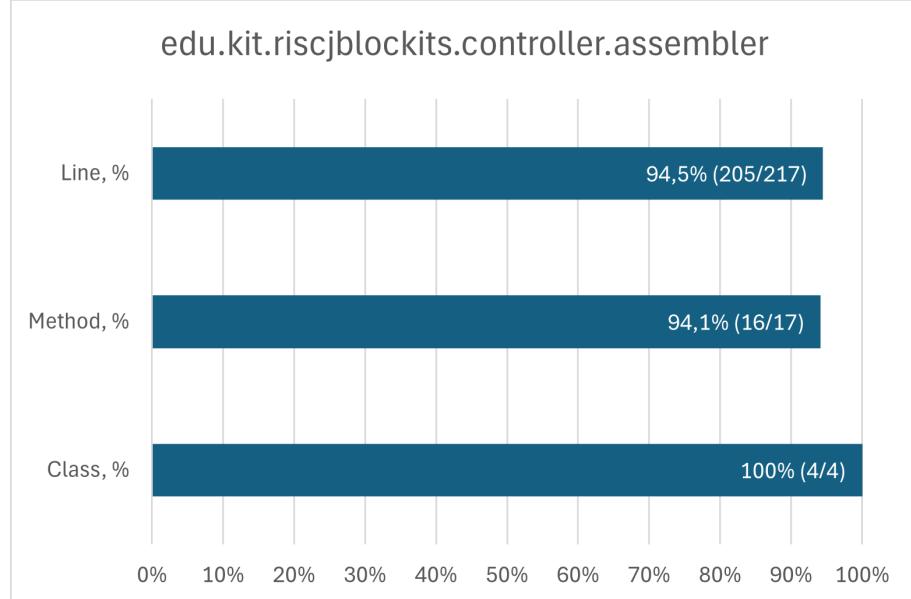


Main.Items

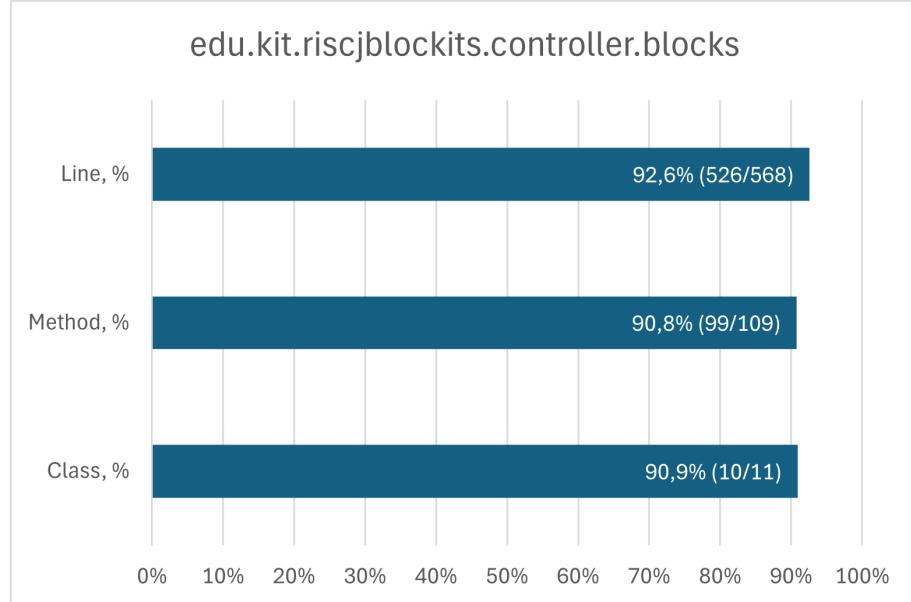


Controller

Assembler

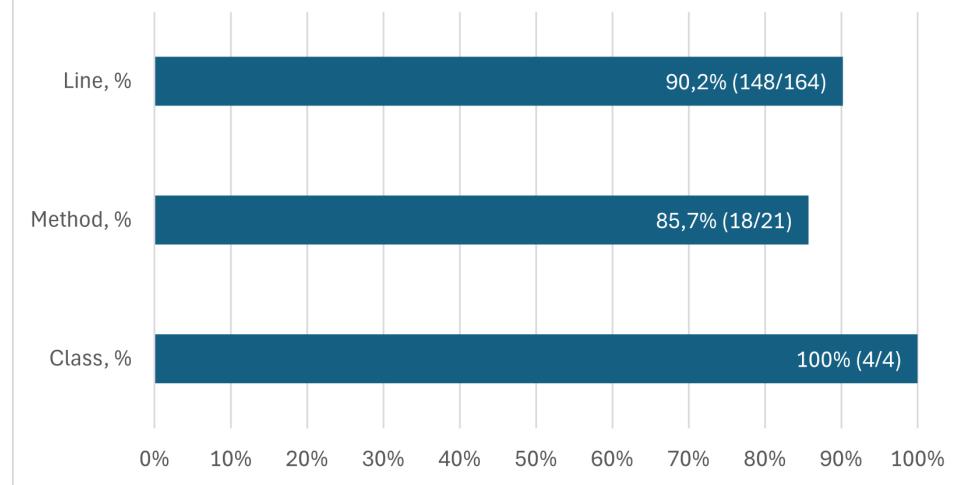


Blocks



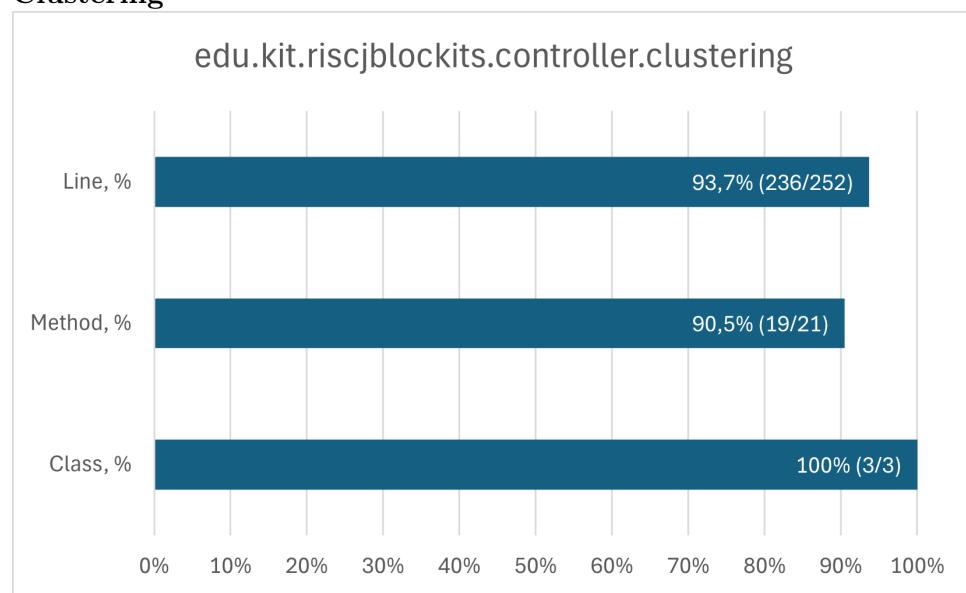
Blocks.IO

edu.kit.riscjblockits.controller.blocks.io

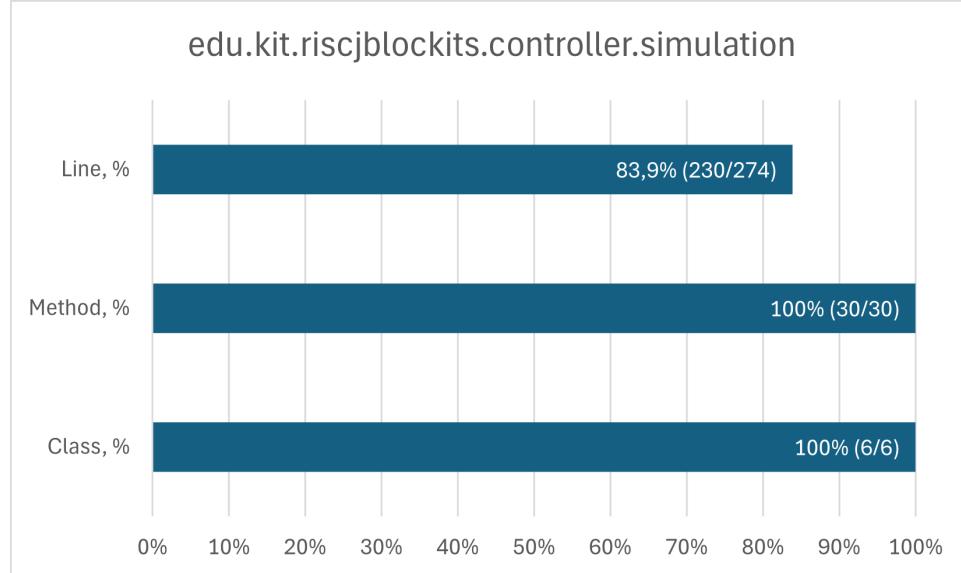


Clustering

edu.kit.riscjblockits.controller.clustering



Simulation



2.2 Integrationstests

Um die Zusammenarbeit verschiedener Komponenten zu testen, wurden abgewandelte Integrationstests durchgeführt. Zunächst wurde geprüft, ob diese mit Minecraft durchführbar sind. Dies war über das Gesamtsystem hinweg allerdings nicht möglich, da es für Minecraft keine Anbindung für Integrationstests gibt. Es wurden einige Test-Frameworks wie der McTester und MinecraftJUnit entdeckt, welche allerdings nur für Forge gebaut sind. Zudem hätte die Möglichkeit bestanden, mit SpecFlow und Mineflayer selbst ein Test-Framework zu bauen. Dies wäre allerdings so zeitintensiv gewesen, dass es den Mehraufwand nicht rechtfertigte. Stattdessen wurden zwei Alternativen gewählt, um so ohne Minecraft Integrationstests durchzuführen.

2.2.1 Große JUnit Tests

Für die Integrationstests wurden mit JUnit5 größere Tests erstellt, die sich über mehrere Komponenten erstreckten. Dafür wurden mithilfe von Mockito alle benötigten Elemente von nicht zu testenden Komponenten als Stubs erstellt.

MIMA bauen

Um zu testen, dass Aktionen im View auch den richtigen Effekt im Controller und Model haben gibt es eine Testreihe, die künstlich in einer gemockten Welt alle Blöcke für einen MIMA Com-

puter platziert und Benutzereingaben direkt künstlich ausführt. Dabei wird getestet, ob Blöcke einen richtigen Status haben und ob sich das Cluster richtig erweitert. Dabei können Server-Client-Kommunikation und Minecraft-Wechselwirkungen zwar nicht getestet werden, sonst wird aber durch diesen Test sichergestellt, dass ein Benutzer einen MIMA-Computer bauen kann.

Simulation-Tests

Um die Zusammenarbeit von Simulation und Model zu testen, sowie zu prüfen, ob das Erstellen von Simulationsthreads funktioniert, wurden Tests für den SimulationTimeHandler und den SimulationSequenceHandler geschrieben. Dabei wurden im SimulationTimeHandler zunächst die drei Möglichkeiten, wie ein Simulationstakt aufgerufen werden kann, getestet. Dabei wurde für onMinecraftTick und onUserTickTrigger geprüft, ob jeweils nur ein Takt ausgeführt wurde. Zudem wurde unterschieden, dass beim MinecraftTick der Taktzähler erhöht wird, beim UserTickTrigger jedoch nicht. Schließlich wurde davon noch der SimulationTickComplete für den Echtzeitmodus abgegrenzt, bei dem nach der Aktivierung gleich mehrere Takte ausgeführt werden und der Taktzähler nicht erhöht wird. Die Unterscheidung der Fälle ist in Abbildung 2.1 dargestellt.

	onMinecraftTick	onUserTickTrigger	onSimulationTickComplete
TickCounter erhöht	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Nur ein Takt ausgeführt	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>

Abbildung 2.1: Unterscheidung der Aufrufarten des SimulationTimeHandlers zur Überprüfung der Korrektheit in Tests

Ein Problem, das bei diesem Test auftrat, war die Verzögerung bei der Erstellung von Threads, welche bei gleichzeitigem Ausführen aller Testfälle dazu führte, dass Methodenaufrufe nicht mehr eindeutig einem Testfall zugeordnet werden konnten und somit *verify* falsche Werte ausgab. Dieses Problem wurde gelöst, indem zwischen Aufruf der zu testenden Methode und Verifikation von darin aufgerufenen Methoden eine kurze Verzögerung durch *sleep* die Zeit für die Thread-Erstellung überbrückte.

2.3 Systemtests mit manuellen Testprotokollen

Für die manuellen Integrationstests gibt es folgend mehrere Protokolle, die alle wichtigen Kriterien aus dem Pflichtenheft abdecken.

2.3.1 MIMA bauen:

Vorbedingungen:

Die Minecraft Java Edition ist gestartet und der Spieler hat eine neue Welt geladen. Die Welt wurde nach folgenden Einstellungen erstellt:

Singleplayer

- > Create New World
- > Namen eingeben
- > Game Mode: Creative
- > Difficulty: Peaceful
- > Allow Cheats: ON (default)
- > World Tab - World Type: Superflat
- > Create New World

Vorgehen:

1. E drücken → Inventar öffnet sich.

2. Auf die zweite Seite wechseln.



3. Items 1-7, 12 und 16 nehmen und in das Inventar schieben.



4. E drücken → Inventar schließt sich.
5. 3 drücken und die Control-Unit auf dem Boden platzieren.
6. Auf den Block schauen und rechts-klicken → ein Fenster öffnet sich.



7. Das Item in Slot 8 nehmen und in das graue Rechteck oben ziehen.



8. ESC drücken → der Screen schließt sich.
9. Auf dem Boden den Aufbau aus 2.2 bauen.



Abbildung 2.2

10. Baue einmal ein Register und einen Bus ab und setzte es wieder.
11. Setze jetzt alle Register auf den Typ, der im Bild auf dem Register steht. Rechts klicke dafür auf einen Registerblock. Klicke dann auf das Buch. Jetzt sind alle möglichen Typen in der Liste. Klicke zum Auswählen und wiederhole das für alle Register.
12. Auf die Control-Unit klicken.



13. Ende des Tests.

Nachbedingungen:

In der Welt steht ein MIMA-Computer nach dem Vorbild von Abbildung 2.2. In die Control-Unit ist ein MIMA-Befehlssatz-Item eingelegt. Alle Register sind zugeordnet. In der Control-Unit wird angezeigt, dass der MIMA-Computer vollständig ist.

Coverage:

Controller: 19%

Model: 30%

View: 32%

2.3.2 MIMA Programm schreiben:

Vorbedingungen:

Die Minecraft Java Edition ist gestartet und der Spieler hat eine Welt geladen. Die Welt wurde nach folgenden Einstellungen erstellt:

Singleplayer

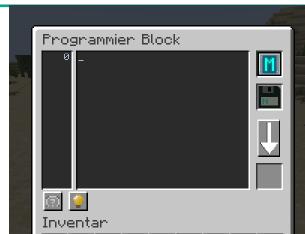
- > Create New World
- > Namen eingeben
- > Game Mode: Creative
- > Difficulty: Peaceful
- > Allow Cheats: ON (default)
- > World Tab - World Type: Superflat
- > Create New World

Vorgehen:

1. E drücken → Inventar öffnet sich.
2. Auf die zweite Seite wechseln.
3. Item 5,12 und 16 in das Inventar ziehen.

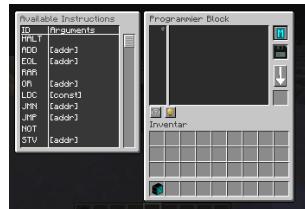


4. Platziere den Programmier-Block auf dem Boden.
5. Mache einen Rechtsklick auf dem Block.
6. Lege das MIMA-Instruction-Set in den obersten Slot.
7. Lege das Programm-Item in den Slot darunter.



8. Drücke den Beispielprogrammknopf (Glühbirnen Icon) → Code erscheint

9. Drücke den Fragezeichen Button → Ein Screen klappt sich aus.



10. Gehe im Text mit dem Cursor in die Zeile 1

11. Lösche die '4' und schreibe eine '10'

12. Gehe mit dem Cursor zur Zeile 11 und füge ein Leerzeichen und ein 'X' ein. → der Text wird rot unterstrichen

13. Drücke fünfmal backspace. → Die Farbe des verbliebenen Textes ändert sich zu grau.

14. Drücke den Beispielprogrammknopf und danach den Assemble Knopf (Pfeil Icon) → Das Programm-Item wandert in den unteren Slot.

15. Nehme das Programm-Item und lege es in den ersten Inventar Slot.

Nachbedingungen:

Auf dem Programm-Item im ersten Inventar-Slot steht das MIMA-Beispielprogramm.

Coverage:

Controller: 11%

Model: 28%

View: 29%

2.3.3 MIMA-Programm im Step-Modus ausführen

Vorbedingungen:

Minecraft ist gestartet und der Spieler hat eine Welt geladen. Test 1 und 2 wurden darin erfolgreich ausgeführt. Die Minecraft-Tageszeit ist Nacht.

Vorgehen:

1. Gehe zum Speicher-Block.
2. Öffne die GUI durch Rechtsklick.
3. Platziere das Programm-Item aus dem ersten Inventar-Slot in das hellgraue Quadrat rechts oben.
4. Scrolle bis zur Speicherstelle 20 und gleiche mit der Abbildung 2.3 ab.



Abbildung 2.3

5. Schließe die GUI durch ESC.
6. Gehe zum System-Clock-Block.
7. Öffnen die GUI durch Rechtsklick.
8. Prüfe, ob oben links 0.0 ticks/s steht und der Zeiger rechts auf den Punkt ganz unten links zeigt.
9. Schließe die GUI durch ESC.
10. Klicke E, um das Inventar zu öffnen.
11. Suche einen Button und platziere diesen auf dem System-Clock-Block.
12. Drücke den Button ein Mal.

13. Gleiche mit Abbildung 2.4 Bild 1 ab.

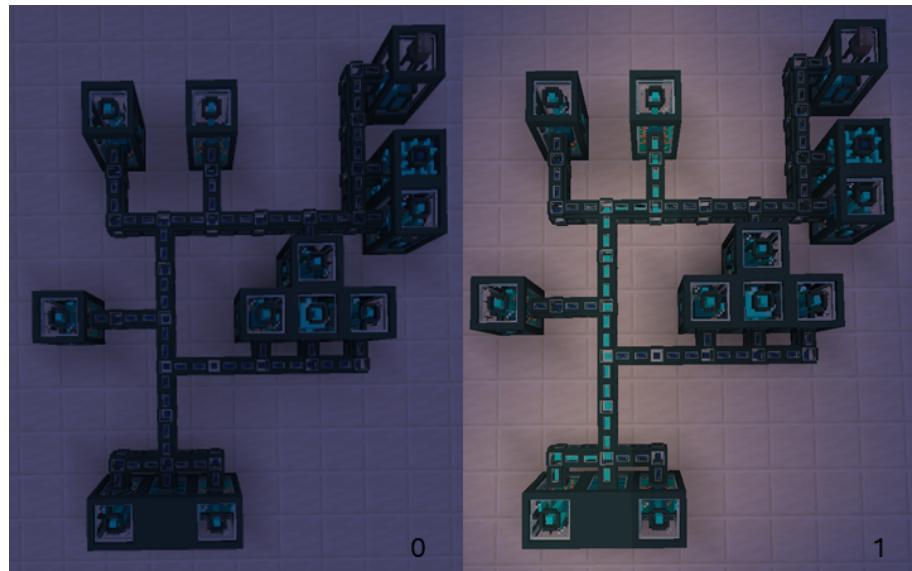


Abbildung 2.4

14. Drücke den Button ein Mal.

15. Gleiche mit Abbildung 2.5 Bild 1,5 ab.

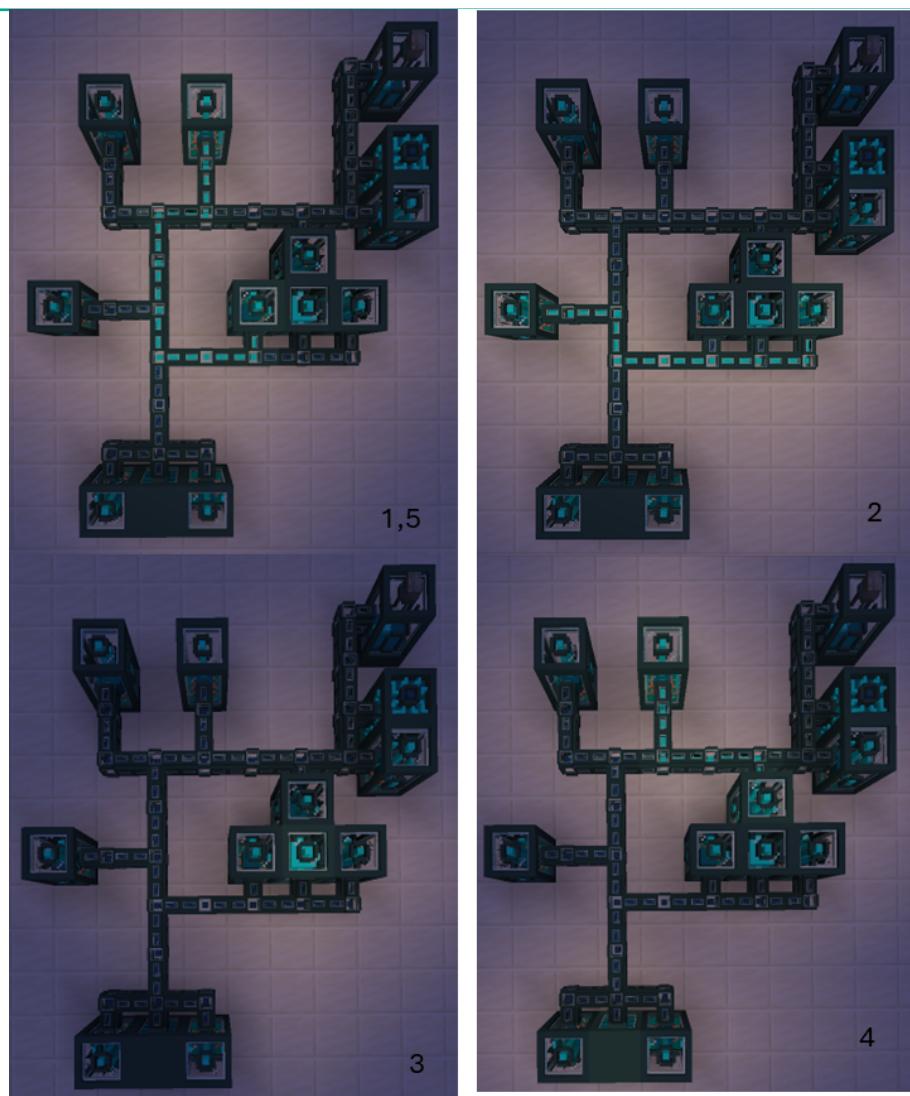


Abbildung 2.5

16. Drücke den Button ein Mal.
17. Gleiche mit Abbildung 2.5 Bild 2 ab.
18. Drücke den Button ein Mal.
19. Gleiche mit Abbildung 2.5 Bild 3 ab.
20. Drücke den Button ein Mal.
21. Gleiche mit Abbildung 2.5 Bild 4 ab.
22. Drücke den Button ein Mal.

23. Gleiche mit Abbildung 2.6 Bild 5 ab.

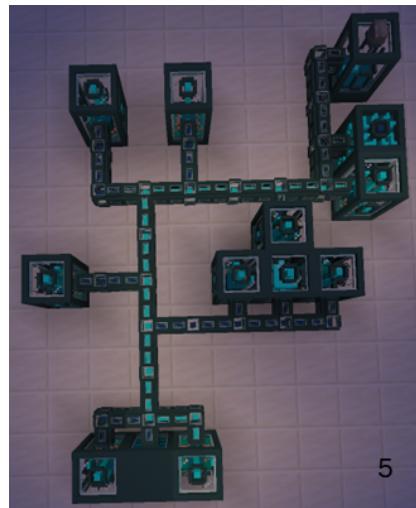


Abbildung 2.6

Nachbedingungen:

Die Fetch-Phase wurde erfolgreich ausgeführt.

Coverage:

Controller: 36%

Model: 56%

View: 24%

2.3.4 Goggle-Test

Vorbedingungen:

Minecraft ist gestartet und der Spieler hat eine Welt geladen. Test 1, 2 und 3 wurde erfolgreich ausgeführt und die Welt nach Test 3 nicht verlassen.

Vorgehen:

1. E drücken → Inventar öffnet sich.

2. Auf die zweite Seite wechseln.

3. Item 15 in das Inventar ziehen.

4. Auf die Truhe unten rechts klicken und die Brille in den Kopf-Slot ziehen.



5. Drücke ESC.
6. Richte das Fadenkreuz auf den Bus:



7. Richte das Fadenkreuz auf das IAR-Register -> Wert: 000021
8. Richte das Fadenkreuz auf das X-Register -> Wert: 000020
9. Richte das Fadenkreuz auf die Alu -> Operation: ADD

Nachbedingungen:

Der Zustand der Welt (abgesehen vom Spieler und dessen Ansicht) hat sich nicht verändert.

Coverage:

Controller: 28%

Model: 41%

View: 20%

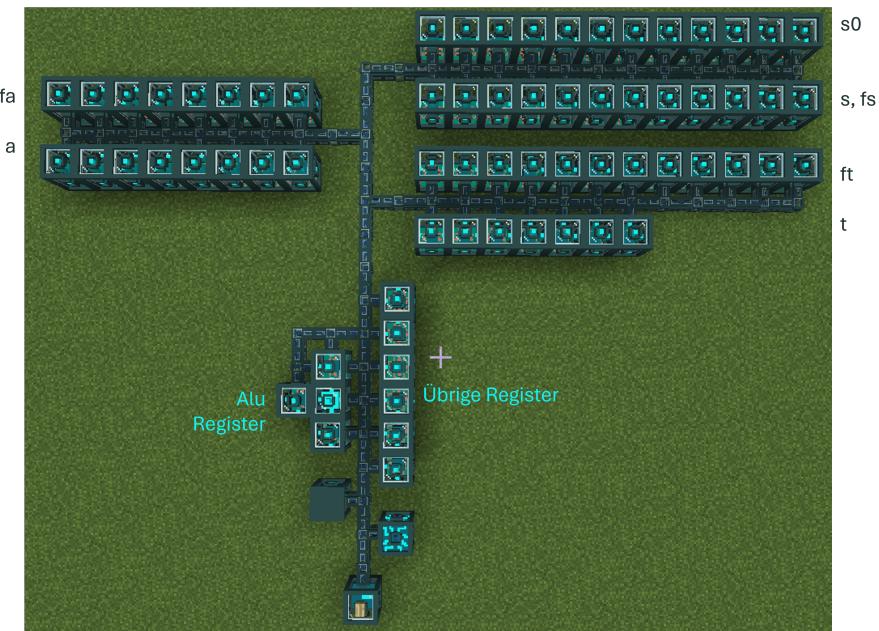
2.3.5 RISC-V-Computer bauen

Vorbedingungen:

Minecraft ist gestartet und der Spieler hat eine Superflat-Welt geladen (vgl. Vorbedingungen der vorigen Tests).

Vorgehen:

1. Baue einen RISC-V-Computer, wie auf dem Bild gezeigt.



2. Setze in die Control-Unit ein RISC-V-Befehlssatz ein.
3. Setze die Registertypen wie im Bild oben beschrieben.

Nachbedingungen:

In der Superflat-Welt steht ein vollständiger RISC-V-Computer.

Coverage:

Controller: 23%

Model: 36%

View: 32%

2.3.6 RISC-V-Beispielprogramm laden und compilieren

Vorbedingungen:

Minecraft ist gestartet und der Spieler hat eine Superflat-Welt geladen (vgl. Vorbedingungen der vorigen Tests).

Vorgehen:

1. Stelle einen Programmierblock auf und lege ein RISC-V-Befehlssatz-Item und ein Programmcode Item in die entsprechenden Slots.
2. Drücke den Beispielprogrammknopf (Glühbirnen Icon).



3. Drücke den Assemble-Knopf (Pfeil-Icon) → Das Programm-Item wandert in den unteren Slot.
4. Nehme das Programm-Item und lege es in den ersten Inventar-Slot.

Nachbedingungen:

Auf dem Programm-Item im ersten Inventar-Slot steht das RISC-V-Beispielprogramm

Coverage:

Controller: 14%

Model: 27%

View: 25%

2.3.7 RISC-V-Programm im Fast-Modus ausführen

Vorbedingungen:

In einer ansonsten leeren Superflat-Welt steht ein korrekter RISC-V-Computer, was sich anhand des Steuerwerk-Blocks prüfen lässt, in dem ein RISC-V-Befehlssatz-Item eingelegt ist. Auf einem

Programm-Item im ersten Inventar-Slot steht das RISC-V-Beispielprogramm.

Vorgehen:

1. Gehe zum Speicherblock und lege das Programm-Item ein. → In der Speicher Übersicht ab Zeile 14 erscheint das Programm.

Memory	
0x	0x
14	00000000
15	00000017
16	00000010
17	00000045
18	00000002
19	00000000
1A	0
1B	0
1C	0
Inventory	

2. Mache einen Rechtsklick auf dem Taktgeberblock und stelle den Fast Mode ein:



3. Drücke den Knopf an dem Taktgeber → Der Computer läuft.
4. Warte, bis der Computer aufhört, zu leuchten.
5. Mache wieder einen Rechtsklick auf den Speicherblock. Scrolle zu Zeile 14 und vergleiche die Speicherübersicht:

Memory	
0x	0x
14	00000000
15	00000002
16	00000010
17	00000017
18	00000045
19	00000080
1A	0
1B	0
1C	0
Inventory	

Abbildung 2.7

Nachbedingungen:

Das Beispielprogramm ist erfolgreich gelaufen. Der Speicher hat den korrekten Endzustand wie in Abbildung 2.7. Der Computer leuchtet nicht mehr und befindet sich im Step-Modus.

Coverage:

Controller: 40%

Model: 62%

View: 25%

3 Nicht-funktionale Tests

3.1 User Tests

Wichtiger Teil der nicht-funktionalen Anforderungen des Projekts war eine intuitive Benutzbarkeit. Zudem verfolgt die Modifikation das Ziel einer Lernanwendung. Um festzustellen, ob diese Ziele erreicht wurden, wurden User Tests durchgeführt.

Die Zahl der Tester liegt bei 24. Diese sind mit großer Mehrheit männlich und haben überwiegend bereits Kenntnisse in der Computerarchitektur und der Programmierung. Bei den meisten liegt auch Erfahrung mit MIMA-, RISC-V- oder Assembler-Programmierung vor. Es ist festzustellen, dass die Tester sehr qualifiziert für die Suche von Fehlern und Bugs im Projekt waren, für die Testung der nicht-funktionalen Anforderungen allerdings keine repräsentative Gruppe darstellen. Nichtsdestotrotz lassen sich auch aus dem Feedback der Tester Rückschlüsse über den Projekterfolg ziehen.

3.1.1 Einige Stimmen unserer Tester

«*boaaah das ist crazy cool gemacht
richtig nices ding was ihr da programmiert habt :D*»
Informatik-Student des KIT im 5. Fachsemester.

«*it [...] looks fantastic, plays decently too and overall is incredibly promising*»
Unbekannter Nutzer von CurseForge

«*Ich finde die Mod ist recht einfach zu verstehen und durch das Handbuch, welchen einen Schrittweise an die einzelnen Komponenten und deren Nutzen heranführt, auch für Anfänger geeignet. Man kann auch durch die Beleuchtung des Bus schön erkennen, wie genau so ein Computer funktioniert. Die Blocktexturen sind sehr gelungen, auch wenn ich teilweise Probleme hatte die einzelnen Blöcke auseinanderzuhalten, da sie durch das gleiche Schema recht ähnlich aussehen. Alles in allem würde ich sagen, dass es eine durchaus gelungene Mod ist, welche mir persönlich auch Spaß bereitet hat, da man spielerisch etwas neues lernen kann.*»
Informatik-Student der DHBW im 3. Fachsemester

3.1.2 Feature Requests der Tester

- sort available instructions in some sort
- Assembler Error invalid line is very unspecific
- make clear what directives are supported
- support pseudo instructions
- Texteditor Benutzerfreundlichkeit erhöhen
- “ARE YOU SURE“-Abfrage vor dem Überschreiben des aktuellen Inhalts des Texteditors durch das Beispielprogramm
- Scrollwheel is not interactive

3.1.3 Von Testern gefundene Bugs

- Texteditor löscht Inhalt, wenn vom Fullscreen- zum Fenstermodus gewechselt wird
- Visual Bug in Terminal
- Item tooltips have no text in the memory menu
- The block selection outline is transparent for most of the mods blocks
- Item quickmove does not work in the control unit menu
- RiscV Instruction Set schließen führt zu IllegalArgumentException
- Bug in Select Register UI
- Block entity data deserialisation throws exceptions in a packet handler causing logspam

3.2 Leistungstests

Um zu prüfen, dass der MIMA-/RISC-V-Computer auch auf verschiedenen Endgeräten zuverlässig und schnell läuft, wurden Leistungstests durchgeführt.

Es wurde auf folgenden Geräten getestet:

- Desktop-PC:
 - Prozessor: Ryzen 9 5900X
 - RAM: 32GB

- Grafikkarte: RX 6700XT
- Laptop:
 - Prozessor: Ryzen 7 3700U
 - RAM: 6GB
 - Grafikkarte: RX Vega 10

3.2.1 MIMA-Leistungstest

Während des MIMA-Leistungstests wurde ein MIMA-Programm, das 8 Millionen MIMA-Assembler-Instruktionen zur Ausführung braucht, im Echtzeitmodus ausgeführt. Dabei wurde die Zeit gestoppt, um die Geschwindigkeit der MIMA-Simulation zu ermitteln.

Die Geräte schnitten folgendermaßen ab:

- PC: 150s -> ca. 53k ausgeführte Instruktionen pro Sekunde
CPU nur auf einzelnen Kernen ausgelastet (CPU-Gesamtauslastung $\leq 10\%$)
2GB (begrenzt durch Minecrafts Standard-RAM-Maximum) von Minecraft genutzter Arbeitsspeicher
- Laptop: 342s -> ca. 23k ausgeführte Instruktionen pro Sekunde
CPU-Auslastung $\geq 90\%$
1,4GB von Minecraft genutzter Arbeitsspeicher

3.2.2 RISC-V-Leistungstest

Während des RISC-V-Leistungstests wurde ein RISC-V-Programm für zwei Minuten im Echtzeitmodus ausgeführt. Danach wurden die Register ausgelesen und damit die Anzahl der ausgeführten Instruktionen ermittelt.

Die Geräte schnitten folgendermaßen ab:

- PC: ca. 39k ausgeführte Instruktionen pro Sekunde
CPU und RAM hatten eine ähnliche Auslastung wie beim MIMA-Leistungstest.
- Laptop: ca. 17,6k ausgeführte Instruktionen pro Sekunde
CPU und RAM hatten eine ähnliche Auslastung wie beim MIMA-Leistungstest.

Die langsamere Ausführung liegt daran, dass RISC-V deutlich mehr Befehle, oft mit dem gleichen op-code, besitzt, unter welchen der Auszuführende zu finden ist.

3.2.3 Speicher-Lasttest

Während des Speicher-Lasttests wurde ein MIMA-Programm, das über alle Speicheradressen iteriert und diese dabei mit Einsen befüllt, im Echtzeitmodus ausgeführt. Dieser Test prüft, ob die Simulation und Speicher-Anzeige im View mit größeren Datenmengen zurechtkommt. Sowohl der Laptop als auch der PC schaffen es dabei, den Lasttest ohne Lags oder Spielabstürze zu durchlaufen. Bei größerem Speicherinhalt nimmt jedoch die Geschwindigkeit der Ausführung ab. Dies wirkt sich allerdings nur auf den Echtzeitmodus aus. Die CPU und der RAM hatten ähnliche Auslastungswerte wie beim MIMA-Leistungstest.

4 Sonstiges

4.1 Verbliebene Bugs

- Quantum State Register does not work in the nether and end#306
- in some small window sizes, the side widget gets rendered ontop the main widget of some screens #298

4.2 Statische Tests

Durch die Integrationsphase und die Qualitätssicherung hinweg wurden Tools zur statischen Codeanalyse eingesetzt. Sehr direktes und schnelles Feedback gab dabei der in der IDE integrierte statische Codechecker von IntelliJ. Für erweiterte Analysen wurde das SonarLint-Plugin verwendet. Beides war sehr wertvoll, um Defekte zu erkennen, bevor sie zu einem Versagen führen konnten.

4.3 Testautomatisierung

Durch alle Phasen, in denen Code produziert wurde, kam Testautomatisierung zum Einsatz. Dafür wurde bei jedem Push auf das GitLab Repository eine GitLab-CI-Pipeline ausgelöst. Sie ist in 3 Stufe unterteilt: In der ersten wird das Projekt kompiliert. In der zweiten werden alle JUnit-Tests ausgeführt und die Überdeckung mit JaCoCo gemessen. In der dritten testet SonarQube das Projekt auf statische Fehler. Ein Discord-Bot gibt schnelle Rückmeldung über Erfolg und Misserfolg.

Dieses Setup hat uns erlaubt, sofort zu bemerken, wenn Änderungen des Codes ungewollte Auswirkungen hatten. Außerdem konnten so Merge-Requests besser eingeschätzt und nur wenn wirklich fertig gemerkt werden.

5 Glossar

Assembler: Ein Assembler ist ein Computerprogramm, das Assemblersprachcode in Maschinencode übersetzt. Assemblersprache ist eine Low-Level-Programmiersprache, die eng mit dem Maschinencode verwandt ist, aber für Menschen leichter zu lesen und zu schreiben ist.¹

Befehlssatz:

Der Befehlssatz eines Prozessors ist in der Rechnerarchitektur die Menge der Maschinenbefehle, die ein bestimmter Prozessor ausführen kann. Je nach Prozessor variiert der Umfang des Befehlssatzes zwischen beispielsweise 33 und über 500 Befehlen. CISC-Prozessoren haben tendenziell größere Befehlssätze als RISC-Prozessoren.²

Block:

Ein Block ist ein besonderer Gegenstand, den man in der Minecraft-Welt platzieren kann. Nahezu die gesamte Spielwelt besteht aus Blöcken.³

Inventar:

Im Überlebensmodus ist das Inventar der Speicher des Spielers für Blöcke und sonstige Gegenstände, sozusagen ein Rucksack, den man immer bei sich trägt. Das Inventar hat ein begrenztes Fassungsvermögen. Im Kreativmodus ist das Inventar kein Speicher, sondern eine Auswahl fast aller Blöcke und sonstiger Gegenstände in unbegrenzter Menge.⁴

¹Quelle: <https://techwatch.de/blog/understanding-the-basics-what-is-an-assembler-and-how-does-it-work/>

²Quelle: <https://de.wikipedia.org/wiki/Befehlssatz>

³Vgl.: <https://minecraft.fandom.com/de/wiki/Block>

⁴Quelle: <https://minecraft.fandom.com/de/wiki/Inventar>

Item:

Ein Gegenstand (engl. Item) ist alles, was man in sein Inventar aufnehmen und in der Hand halten kann. Ein Block ist ein besonderer Gegenstand, den man in der Welt platzieren kann (z.B. Erde oder eine Werkbank). Daneben gibt es noch einige Gegenstände, die beim Platzieren zu einem beweglichen Objekt werden, z.B. ein Boot.⁵

MIMA:

Die mikroprogrammierte Minimalmaschine (MIMA) ist ein Lehrmodell zur vereinfachten Darstellung von Mikroprozessoren, basierend auf der Von-Neumann-Architektur, welche von Taimim Asfour am Karlsruher Institut für Technologie entwickelt wurde.⁶

Minecraft-Objekte:

Objekte (engl. Entities) sind neben den Gegenständen, zu denen auch die Blöcke gehören, die andere große Gruppe der Spielelemente in Minecraft. Eine Minecraft-Welt besteht aus Blöcken und Objekten. Im Gegensatz zu den Blöcken sind die Objekte meist beweglich (z.B. eine Lore oder Projektil). Zu den Objekten zählen auch jegliche Drops.⁷

Mod:

Als Modifikation (Abkürzung Mod) wird alles bezeichnet, das den Spielinhalt von Minecraft verändert.⁸

Redstone:

Redstone ist ein flacher, transparenter Block, der Redstone-Signale übertragen kann. Ein Signal geht von einem Signalblock über den Redstone zu einem Empfängerblock und löst dort eine Aktion aus.

RISC-V:

RISC-V ist eine Befehlssatzarchitektur, die sich auf das Designprinzip des Reduced Instruction Set Computers (RISC) stützt. Das Designziel von RISC ist der Verzicht auf einen komplexen Befehlssatz hin zu einfach zu dekodierenden und schnell auszuführenden Befehlen.⁹

Slot:

Ein Slot in Minecraft ist ein Platz in der GUI, an den Items gelegt werden können. Das Inventar und Kisten bestehen aus Slots.

⁵Quelle: <https://minecraft.fandom.com/de/wiki/Gegenstand>

⁶Vgl.: https://de.wikipedia.org/wiki/Mikroprogrammierte_Minimalmaschine

⁷Vgl.: <https://minecraft.fandom.com/de/wiki/Objekt>

⁸Quelle: <https://minecraft.fandom.com/de/wiki/Modifikation>

⁹vgl.: <https://de.wikipedia.org/wiki/RISC-V> und https://de.wikipedia.org/wiki/Reduced_Instruction_Set_Computer