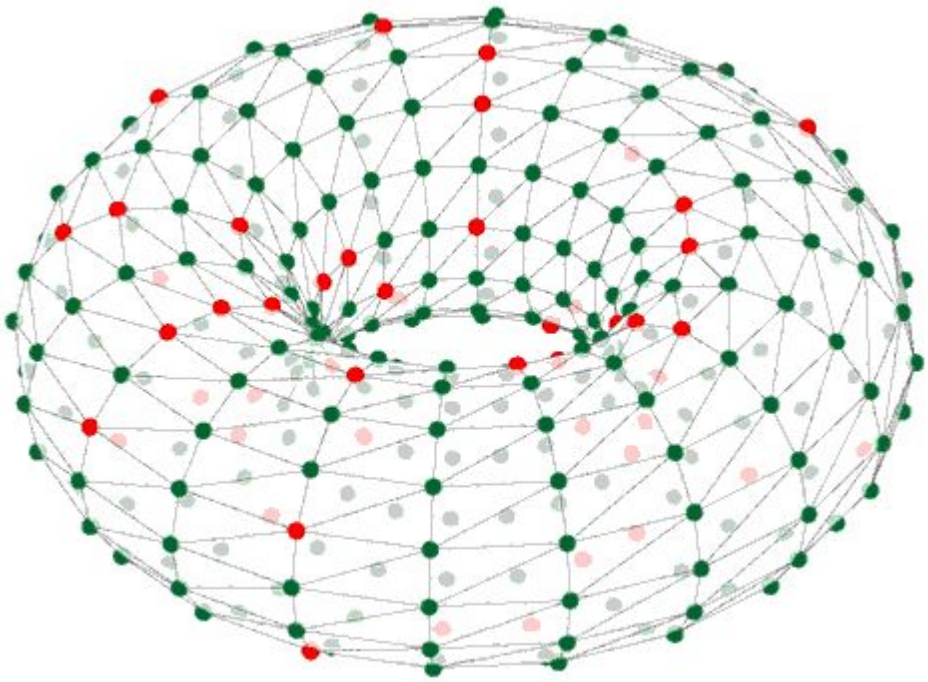
5th SpiNNaker Workshop

Proceedings

September 7th- 12th 2015



Manchester, UK

Fifth SpiNNaker Workshop

School of Computer Science, Manchester, UK, 7th – 11th September 2015

Day 1: Introductory course **Monday 7th September 2015**

Time	Session	Presenter(s)
12:00	Lunch and Registration	
13:00	Workshop logistics	Simon
13:15	SpiNNaker Hardware & Tools Overview	Simon
14:00	Running PyNN simulations on SpiNNaker	Andrew
15:00	Coffee	
15:30	Lab time	
17:00	Close	

Day 2: Introductory course **Tuesday 8th September 2015**

	Session	Owner
09:00	Synaptic plasticity using PyNN	Sergio
10:00	Lab time (with coffee at 10:30)	
12:00	Lunch	
13:00	Simple data I/O and visualisation	Alan
14:00	Lab time (with coffee at 15:00)	
17:00	Close	

Day 3: Introductory course **Wednesday 9th September 2015**

Time	Session	Owner
09:00	Free lab time (with coffee at 10:30)	
12:00	Lunch and close	

Note: Last session of Beginner's Workshop overlaps with first session of the Advanced Workshop.

Day 3: Advanced course **Wednesday 9th September 2015**

Time	Session	Owner
09:00	Registration and coffee	
09:15	SpiNNaker architecture, chip resources and limitations	Steve
10:00	Adding new neuron models	Andrew/Michael
10:30	Coffee	
11:00	Lab time	
12:00	Lunch	
13:00	SpiNNaker system software (SARK)	Steve
13:45	SpiNNaker API	Luis
14:30	Lab time (with coffee at 15:00)	
16:00	Event-driven neural simulation	Alex
16:30	Lab time	
17:00	Close	

Day 4: Advanced course **Thursday 10th September 2015**

Time	Session	Owner
09:00	Maths & fixed point libraries	Michael
09:45	Adding new models of synaptic plasticity	Jaime
10:30	Lab time (with coffee from 10:30)	
12:00	Lunch	
13:00	Connecting to external devices	Alex/Sergio
13:45	Lab time (with coffee at 15:00)	
16:00	Debugging using YBUG & GDB	Steve
17:00	Close	

Day 5: Advanced course **Friday 11th September 2015**

Time	Session	Owner
09:00	Free lab time	
10:30	Coffee	
11:00	Free lab time	
12:00	Lunch and close	

5th SpiNNaker Workshop – 7th – 11th September, 2015

Dear Workshop Participant,

Here is some logistics information to help you to plan your visit to Manchester for the SpiNNaker workshop in September. If at any point you find that you are unable to attend, just let us know.

Food

We provide lunch and refreshments at the workshop but leave you to make your own arrangements in the evening. If you have any specific dietary requirements, let us know as soon as possible.

Workshop Times

We start at 9am on Monday 7th Sept for the basic workshop and 9am on Wednesday 9th Sept for the advanced workshop. We ask participants to make every effort to arrive on time as the early sessions are often the most information-dense!

Each workshop 'day' goes onto 5pm, but the late afternoon sessions contain labs so it would be possible to leave early, if required.

Travel

By air: Manchester airport is a 30 minute taxi ride away from us (more at peak times). This will cost about £20.

By train: Manchester has three stations. Oxford Road is most convenient (5 mins walk from our building). Picadilly station is the main terminal for trains from the south and is 15 min walk from us. Victoria station is north of the centre and you would need a take a taxi to get to us.

Accommodation

The follow hotels in our area have been used by participants in the past and we haven't had too many bad reports:

Holiday Inn Express (5 mins walk up Oxford road)

http://www.hiemanchester.co.uk/

Premier Inn, Deansgate Locks (10 mins walk, near Oxford Road station and town)

http://www.premierinn.com/en/hotel/MANPTI/manchester-city-centre-deansgate-locks

Manchester Conference Centre (10 mins walk towards Picadilly)

http://www.manchesterconferencecentre.co.uk/

We expect participants to arrange their own accommodation.

Finding Our Building

The workshop will be held in the School of Computer Science at the University of Manchester, UK.

This is located in the Kilburn building on Oxford Road and I enclose a campus map to help you find us. The building is numbered 39 on the plan.

Finding us inside the Kilburn Building

The workshop will be based in *collab* 0, located near Bytes Café on the LF1 floor of the Kilburn building and is part of the School of Computer Science at Manchester University. If you enter the Kilburn building by the ground floor, follow the arrows and go up the stairs in front of you. The room is directly ahead, through glass windows. Lunch and refreshments will be served in the open area outside of *collab* 0.

Workshop Pre-requisites

We assume that you will bring your own laptop and that you can plug an Ethernet-based device into it (the SpiNNaker boards connect via Ethernet). From past experience, we know this can be an issue for Mac users, as the Ethernet port may be an optional plugin module.

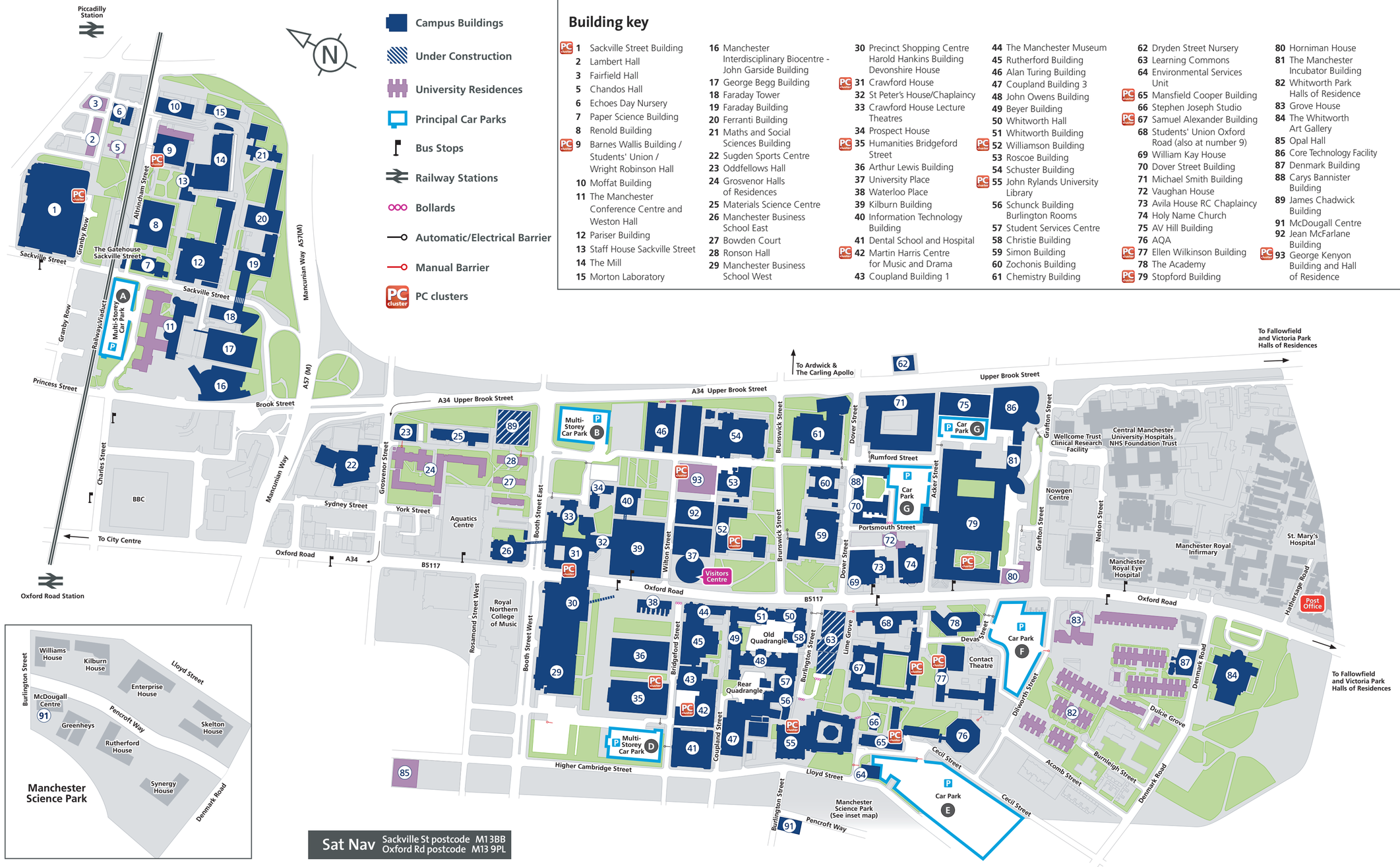
Some knowledge of **Python** is assumed so if you are not already familiar with it, we suggest that you learn the basics before you arrive. For the advanced workshop, some knowledge of **C** is also very useful.

Tools install

To speed up the workshop itself, we ask that you try to install the tools before you arrive. We'll send you a link to enable you to do this. Don't worry if you have any issues, we can sort those out when you are here.

Any Questions?

Let us know.



Building key

1 Sackville Street Building (PC cluster)
2 Lambert Hall
3 Fairfield Hall
5 Chandos Hall
6 Echoes Day Nursery
7 Paper Science Building
8 Renold Building
9 Barnes Wallis Building / Students' Union / Wright Robinson Hall (PC cluster)
10 Moffat Building
11 The Manchester Conference Centre and Weston Hall
12 Pariser Building
13 Staff House Sackville Street
14 The Mill
15 Morton Laboratory
16 Manchester Interdisciplinary Biocentre - John Garside Building
17 George Begg Building
18 Faraday Tower
19 Faraday Building
20 Ferranti Building
21 Maths and Social Sciences Building
22 Sugden Sports Centre
23 Oddfellows Hall
24 Grosvenor Halls of Residences
25 Materials Science Centre
26 Manchester Business School East
27 Bowden Court
28 Ronson Hall
29 Manchester Business School West
30 Precinct Shopping Centre Harold Hankins Building Devonshire House
31 Crawford House (PC cluster)
32 St Peter's House/Chaplaincy
33 Crawford House Lecture Theatres
34 Prospect House
35 Humanities Bridgeford Street (PC cluster)
36 Arthur Lewis Building
37 University Place
38 Waterloo Place
39 Kilburn Building
40 Information Technology Building
41 Dental School and Hospital
42 Martin Harris Centre for Music and Drama (PC cluster)
43 Coupland Building 1
44 The Manchester Museum
45 Rutherford Building
46 Alan Turing Building
47 Coupland Building 3
48 John Owens Building
49 Beyer Building
50 Whitworth Hall
51 Whitworth Building
52 Williamson Building (PC cluster)
53 Roscoe Building
54 Schuster Building
55 John Rylands University Library (PC cluster)
56 Schunck Building Burlington Rooms
57 Student Services Centre
58 Christie Building
59 Simon Building
60 Zochonis Building
61 Chemistry Building
62 Dryden Street Nursery
63 Learning Commons
64 Environmental Services Unit
65 Mansfield Cooper Building (PC cluster)
66 Stephen Joseph Studio
67 Samuel Alexander Building (PC cluster)
68 Students' Union Oxford Road (also at number 9)
69 William Kay House
70 Dover Street Building
71 Michael Smith Building
72 Vaughan House
73 Avila House RC Chaplaincy
74 Holy Name Church
75 AV Hill Building
76 AQA
77 Ellen Wilkinson Building (PC cluster)
78 The Academy
79 Stopford Building (PC cluster)
80 Horniman House
81 The Manchester Incubator Building
82 Whitworth Park Halls of Residence
83 Grove House
84 The Whitworth Art Gallery
85 Opal Hall
86 Core Technology Facility
87 Denmark Building
88 Carys Bannister Building
89 James Chadwick Building
91 McDougall Centre
92 Jean McFarlane Building
93 George Kenyon Building and Hall of Residence (PC cluster)

Sat Nav Sackville St postcode M1 3BB
Oxford Rd postcode M13 9PL

A
78 Academy
32 Access Summit Disability Resource Centre
37 Accommodation Office
31 Accounting and Finance
1 Aerospace Research Centre (UMARI)
60 Age and Cognitive Performance Research Centre
1 Ahmed Iqbal Ullah Race Relations Resource Centre
46 Alan Turing Building
36 Applied Social Sciences
76 AQA (formerly NEAB)
79 ARC Epidemiology Unit
65 Archaeology
35 Architecture and Planning
65 Art History and Visual Studies
36 Arthur Lewis Building
48 Association of University Administrators (AUA)
46 Astrophysics
69 Athletic Union
75 AV Hill Building
73 Avila House (RC Chaplaincy)

B
9 Barnes Wallis Building (PC cluster)
49 Beyer Building
1 Bindery (Joule Library)
14 Biomolecular Sciences
54 Blackett Lecture Theatre
27 Bowden Court
54 Braddick Library
54 Bragg Lecture Theatre
35 Brookes World Poverty Institute
56 Burlington Rooms
56 Burlington Society

C
D Car Park Permit Office (Booth St West)
31 Careers and Employability Division (via Booth St East)
8 Careers Metro
88 Carys Bannister Building
35 Cathie Marsh Centre for Census and Survey Research (CCSR)
34 Central Services Unit
77 Centre for Continuing Education
77 Centre for Educational Leadership
77 Centre for English Language Studies in Education (CELSE)
77 Centre for Occupational and Environmental Health
79 Centre for Integrated Genomic Medical Research (CIGMR)
59 Centre for the History of Science, Technology and Medicine (CHSTM)
65 Centre for New Writing
1 Centre for Manufacture
1 Centre for Process and Integration
5 Chandos Hall
14 Chemical Engineering and Analytical Sciences *(see also building 89)*
61 Chemistry Building
67 Chinese Studies
58 Christie Building/Bistro
12 Civil and Construction Engineering
67 Classics and Ancient History
67 Combined Studies
48 Communications, Media and Public Relations
39 Computer Science
13 Conference Office
67 Confucius Institute
35 Cordingley Lecture Theatre
86 Core Technology Facility
42 Cosmo Rodewald Concert Hall
1 Council Chamber (Sackville Street)
51 Council Chamber (Whitworth Building)
68 Council Chamber (Students' Union)
31 Counselling Service
43 Coupland Building 1
47 Coupland Building 3
31 Crawford House
33 Crawford House Lecture Theatres
30 CRIC (ESRC Centre for Research and Innovation and Competition)
48 Curriculum Innovation

D
12 Dalton Institute
87 Denmark Road Building
41 Dental School and Hospital
47 Dental School Annexe
58 Development and Alumni Relations
30 Devonshire House
79 Diagnostic Radiology
37 Disability Support Office
70 Dover Street Building
42 Drama
62 Dryden Street Day Nursery

E
52 Earth Sciences
67 East Asian Studies
6 Echoes Day Nursery
1 Electrical and Electronic Engineering
20 E&EE High Voltage
36 Economics
29 Eddie Davies Library
77 Education
77 Ellen Wilkinson Building (PC cluster)
1 Engineering and Physical Sciences Faculty Office
1 Engineering Doctorate Centre
67 English and American Studies
23 English Language Centre
36 Environment and Development
64 Environmental Services Unit
1 EPS Foundation Studies
1 Equality and Diversity Office
49 Estates Directorate
70 Eurolens Research
67 European Studies

F
3 Fairfield Hall
19 Faraday Building
18 Faraday Tower
20 Ferranti Building
48 Finance Office
67 French Studies

G
36 Geography
52 Geology
17 George Begg Building
93 George Kenyon Building and Hall of Residence (PC cluster)
67 German Studies
67 Goethe Institute
36 Government
35 Graduate Development Scheme
36 Granada Centre for Visual Anthropology
35 Graphics Support Workshop
65 GM Archaeological Unit
52 GM Geological Unit
83 Grove House
24 Grosvenor Halls of Residences

H
30 Harold Hankins Building
38 Health and Safety Services
34 Higher Education Careers Service Unit
67 History
74 Holy Name RC Church
80 Horniman House
77 Human Communication and Deafness
48 Human Resources
35 Humanities Bridgeford Street (PC cluster)
30 Humanities Faculty Office

I
48 Income Office
40 Information Technology Building
36 Institute for Development Policy and Management (IDPM)
35 Institute for Political and Economic Governance (IPEG)
52 Institute of Health Sciences
53 International Centre for Labour Studies
45 International Development
69 International Society
67 Italian Studies

J
89 James Chadwick Building *(see also building 14)*
38 Japan Centre
67 Japanese Studies
92 Jean McFarlane Building
16 John Garside Building
48 John Owens Building
55 John Rylands University Library (PC cluster)
42 John Thaw Studio
12 Joule Centre
1 Joule Library (Granby Row entrance) (PC cluster)

K
44 Kanaris Lecture Theatre
35 Kantorowich Library
51 Keepers Room
48 Ken Kitchen Committee Room
39 Kilburn Building
50 Knowles Committee Room

L
2 Lambert Hall
23 Language Centre (Oddfellows Hall)
67 Language Centre (Samuel Alexander Building)
52 Law
67 Leamington Rooms
63 Learning Commons
30 Legal Advice Centre
42 Lenagan Library
71 Life Sciences
79 Life Sciences
79 Life Sciences Faculty Office
67 Linguistics

M
77 Management and Leisure
26 Manchester Business School East
29 Manchester Business School West
30 Manchester Centre for Healthcare Management
39 Manchester Computing
11 Manchester Conference Centre
41 Manchester Dental Education Centre (MANDEC)
81 The Manchester Incubator Building/UMIC
30 Manchester Institute of Innovation Research
31 Manchester Institute for the Deaf
16 Manchester Interdisciplinary Biocentre - (John Garside Building)
44 Manchester Museum
29 Manchester Enterprise Centre
110 Manchester Student Homes
47 Manchester University Press
39 Manchester Visualisation Centre
65 Mansfield Cooper Building (PC cluster)
42 Martin Harris Centre for Music and Drama, and Electro Acoustic Studio (PC cluster)
25 Materials Science Centre
46 Mathematics
21 Maths and Social Science Building
91 McDougall Centre
17 Mechanical, Aerospace and Civil Engineering (MACE)
35 Media Centre
59 Medical and Human Sciences Faculty Office
79 Medicine
79 Medical School
71 Michael Smith Building
67 Middle Eastern Studies
67 Modern Languages
10 Moffat Building
15 Morton Laboratory
54 Moseley Lecture Theatre
42 Music
1 Muslim Prayer Hall North
91 Muslim Prayer Hall South

N
36 Manchester eResearch Centre (MeRC)
52 National Primary Care Research and Development Centre (NPCRDC)
42 Novars Research Centre
92 Nursing, Midwifery and Social Work

O
38 Occupational Health
14 Occupational Health (Sackville Street)
23 Oddfellows Hall
85 Opal Hall
88 Optometry Clinics
48 Owens Committee Room

P
7 Paper Science
12 Pariser Building
43 Pear Lecture Theatre
79 Pharmacy and Pharmaceutical Sciences
36 Philosophy
46 Photon Science Research Institute
54 Physics and Astronomy
36 Planning and Landscape
70 Police Office
36 Politics
56 Postgraduate Society
31 Precinct Library, Crawford House (PC cluster)
30 Precinct Shopping Centre
48 President and Vice Chancellor's Offices
30 PREST
40 Prospects House
59 Psychiatry
43 Psychology

R
48 Regional and Economic Affairs
48 Registrar and Secretary
67 Religions and Theology
8 Renold Building
58 Research Office
28 Ronson Hall
53 Roscoe Building
67 Russian Studies
45 Rutherford Building
54 Rutherford Lecture Theatre

S
32 St Peter's Chaplaincy
1 Sackville Street Building
67 Samuel Alexander Building (PC cluster)
56 Schunck Building
54 Schuster Building
59 Simon Building
36 Social Anthropology
37 Social Work
36 Sociology
67 Spanish and Portuguese Studies
77 Speech Pathology
69 Sports Directorate
13 Staff House Sackville Street
35 Staff Training and Development Unit
66 Stephen Joseph Studio
79 Stopford Building (PC cluster)
45 Student Experience Directorate
37 Student Guidance Service
38 Student Health Centre
48 Student Records
1 Student Services Centre Satellite (Joule Library)
57 Student Services Centre
37 Student Support and Services
9 Students' Union Sackville St
68 Students' Union Oxford Rd
22 Sugden Sports Centre

T
48 Teaching, Learning and Support Office
1 Textiles
14 The Mill

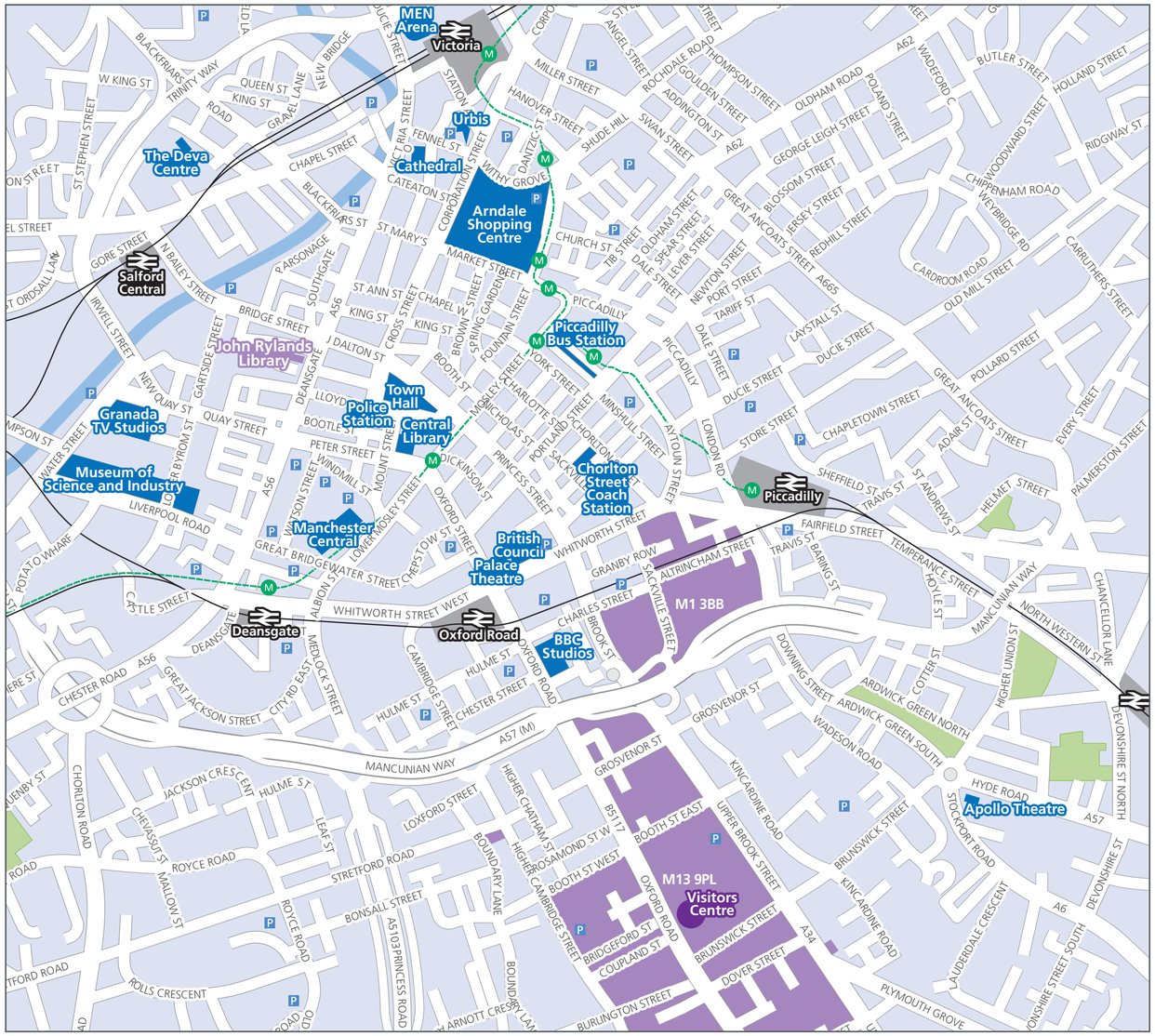
U
1 UMARI
81 UMIC
86 UMIP
52 University College Union
37 University Place

V
72 Vaughan House
37 Visitors Centre
88 Vision Science

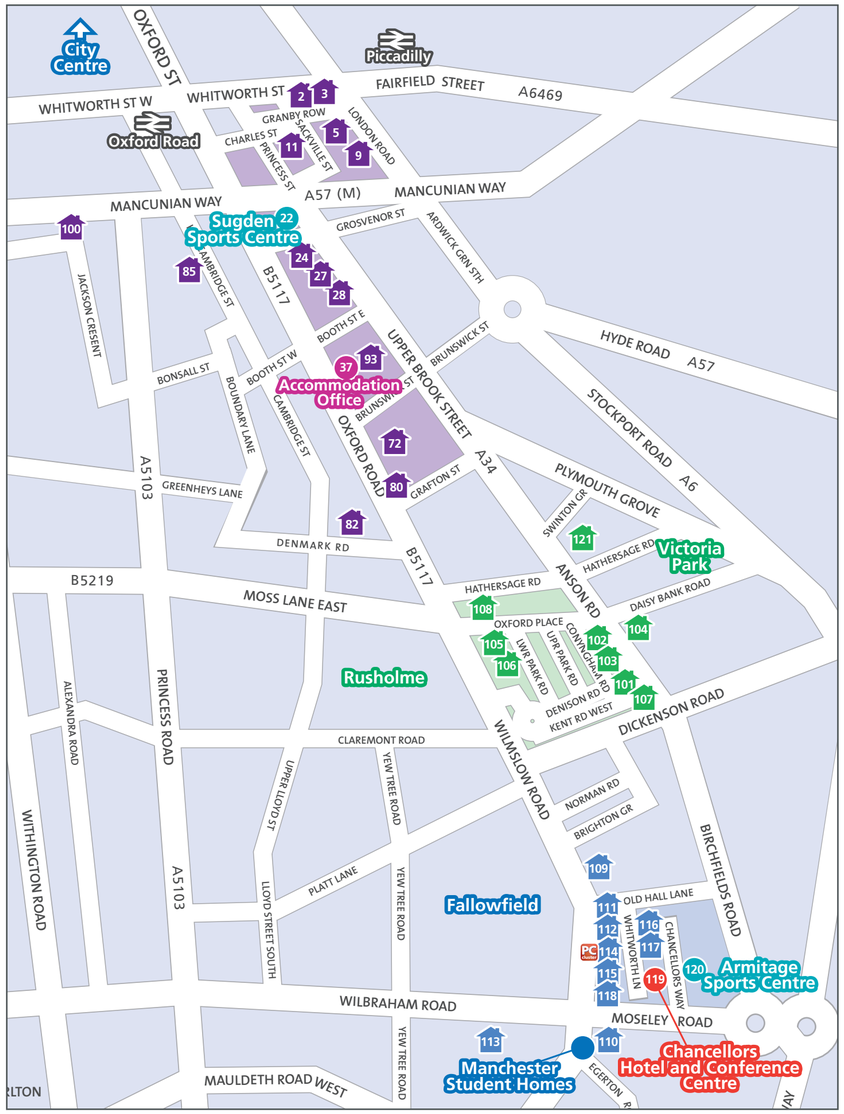
W
38 Waterloo Place
70 Wellbeing Room
11 Weston Hall
84 Whitworth Art Gallery
51 Whitworth Building
50 Whitworth Corridor
50 Whitworth Hall
82 Whitworth Park Halls of Residence
69 William Kay House
52 Williamson Building (PC cluster)
31 Workers' Educational Association
9 Wright Robinson Hall

Z
60 Zochonis Building

City map



Accommodation map



Building key

37	Accommodation Office
109	Allen Hall
120	Armitage Sports Centre
111	Ashburne Hall
27	Bowden Court
100	Brian Redhead Court
106	Burkhardt House at Hulme Hall
101	Canterbury Court
119	Chancellors Hotel and Conference Centre
5	Chandos Hall
102	Dalton Ellis Hall
3	Fairfield Hall
117	Firs Villa
93	George Kenyon Hall
24	Grosvenor Group of Halls
80	Horniman House
105	Hulme Hall
110	Ladybarn House
2	Lambert Hall
113	Linton House
110	Manchester Student Homes
115	Oak House
85	Opal Hall
104	Opal Gardens
PC 114	Owens Park
103	Pankhurst Court
116	Richmond Park
28	Ronson Hall
112	Sheavyn House
22	Sugden Sports Centre
107	St Anselm Hall
108	St Gabriel's Hall
72	Vaughan House
121	Victoria Hall
11	Weston Hall
82	Whitworth Park
118	Woolton Hall
9	Wright Robinson Hall

Sat Nav

Fallowfield postcode M14 6HD
Victoria Park postcode M14 5ES

Directions to the University

Sat Nav Sackville St postcode M13 3BB
Oxford Rd postcode M13 9PL

By air

Manchester Airport is approximately 10 miles from the University. The taxi fare from Manchester Airport is approximately £20.

By bus

From Piccadilly Train Station catch the 147.
From Piccadilly Bus Station catch any of the following:
14, 16, 41, 42, 43, 44, 48, 111, 140, 142, 157 and 250

From Victoria Train Station, catch the Tram to Piccadilly Bus Station and catch one of above services.

From Manchester Airport catch the 43 bus.

By road

By road to the University Visitors Centre
All approach routes are clearly signposted 'Universities'.

M62 (Eastbound), M602
Leave the M62 at J12 and join the M602. At the end of M602, join Regent Road (A57) and continue along and join the A57M (Mancunian Way). Leave at the second exit sign-posted A34 Congleton (hair-pin bend) and keep left, following signs for the Universities. Turn left onto the A34 (dual carriageway) and get in the right hand lane. Turn right at the first set of traffic lights into Grosvenor Street. Stay in the left hand lane and turn left at the next set of traffic lights onto Oxford Road (B5117). Go straight on through the next set of traffic lights. The University's Visitors Centre is in the distinctive round building on the left-hand side.

M62 (Westbound), M60
Leave M62 at J18 and join M60 ring road. Leave M60 at J22 and turn right at traffic lights onto Oldham Road (A62). Continue along until the end and turn left to join Great Ancoats Street (A665). Follow signs for the Universities and join the Mancunian Way (A57M). Leave at the second exit onto Higher Cambridge Street. Turn left at the next set of traffic lights onto Booth Street West and then turn right onto Oxford Road (B5117). The University's Visitors Centre is in the distinctive round building on the left-hand side.

M6/M56
Leave the M6 at Junction 19 and turn right onto the A556. Follow signs for Manchester Airport and join the M56. Stay on M56 until it joins Princess Park Way (A5103). Continue for a further four miles before turning right at the Royal Brewery into Moss Lane East (B5219). Continue to the T-junction and turn left onto Oxford Road (B5117). Proceed down Oxford Road past the hospitals. The University's Visitors Centre is in the distinctive round building on the right-hand side.

From M67
At the end of the motorway, join Hyde Road (A57). Continue along for approximately three miles, following signs for the City Centre. Upon reaching a major roundabout (the Apollo Theatre is on the left) take the second exit, Brunswick Street (signposted 'Universities'). Continue along and go straight through the next set of traffic lights. At the next set of traffic lights, turn right onto Oxford Rd (B5117). The University's Visitors Centre is in the distinctive round building on the right-hand side.

Car parks

A Charles Street Multi-Storey
Monday to Friday

0-3 hours	£3.00
3-6 hours	£5.00
6-10 hours	£8.00
10-24 hours	£10.00
4pm - 12midnight	£2.00

Saturday & Sunday
All Day £2.00

B Booth St East Multi-Storey

0-2 hours	£2.40
2-4 hours	£4.10
4-9 hours	£5.80
9-24 hours	£8.60

D Booth St West Multi-Storey
Monday to Friday

0-3 hours	£3.00
3-6 hours	£5.00
6-10 hours	£8.00
10-24 hours	£10.00
4pm - midnight	£2.00

Saturday & Sunday
All Day £2.00

D Booth St West (Surface Car Park)
Permit holders only. No visitor parking

E Cecil Street
Monday to Friday
All Day £3.00
Saturday & Sunday
All Day £2.00

F Dilworth Street
Monday to Friday
Daytime - Permit holders only.
Visitor parking
4pm - 12midnight £2.00
Saturday & Sunday
All Day £2.00

G Dover Street
Monday to Friday
Daytime - Permit Holders Only.
Visitor parking
4pm - 12midnight £2.00
Saturday & Sunday
All Day £2.00

Car Park charges correct at time of going to print

Useful contacts

UNIVERSITY CONTACTS

Switchboard
0161 306 6000

Faculty of Engineering and Physical Sciences
0161 306 9100

Faculty of Humanities
0161 306 1100

Faculty of Life Sciences
0161 306 7100

Faculty of Medical and Human Sciences
0161 306 0100

The Academy
0161 275 2930

Chancellors Hotel and Conference Centre
0161 907 7414

Development and Alumni Relations
0161 306 3066

International Development
0161 306 6234

The John Rylands University Library
0161 275 3751

The John Rylands Library Deansgate
0161 306 0555

Manchester Conference Centre
0161 955 8000

The Manchester Museum
0161 275 2634

Security
0161 306 9966

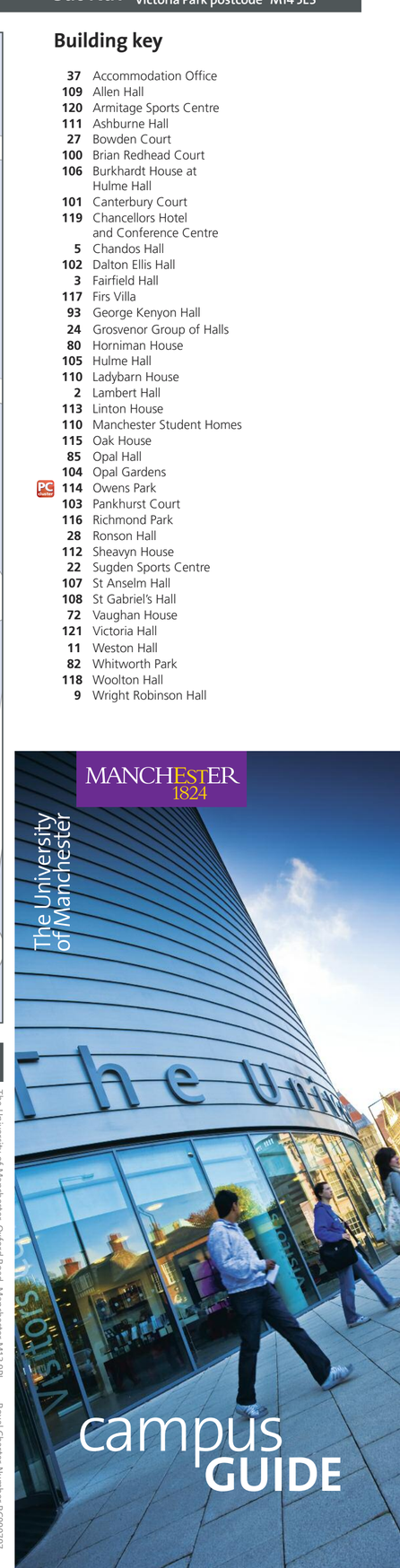
Student Accommodation
0161 275 2888

Students' Union
0161 275 2931

Visitors Centre
0161 275 2922

The Whitworth Art Gallery
0161 275 7450

The University of Manchester, Oxford Road, Manchester M13 9PL
Royal Charter Number RC000797
13375 09.11 V1



5th SpiNNaker Workshop

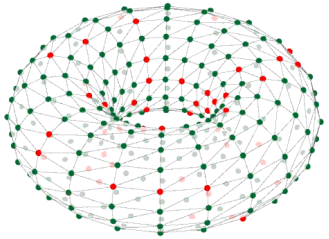
Day 1

September 7th 2015

Time	Session	Presenter(s)
12:00	Lunch and Registration	
13:00	Workshop logistics	Simon
13:15	SpiNNaker Hardware & Tools Overview	Simon
14:00	Running PyNN simulations on SpiNNaker	Andrew
15:00	Coffee	
15:30	Lab time	
17:00	Close	

Manchester, UK

5th SpiNNaker Workshop



Welcome & Overview

SpiNNaker Workshop
September 2015

European Research Council
Established by the European Commission

Human Brain Project

EPSRC

SpiNNaker

Sessions and Venues

- Two types of sessions:
  - Presentations (some optional!)
  - Lab work (with lab books)
- Three venues:
  - ***Collab 1*** (here) for labs and some presentations
  - ***LF 15*** (30m away) for some presentations
  - Area outside this room for lunch/drinks breaks
- Don't leave valuables here overnight!

Breakdown of each day

- Sessions:
  - Start at 9am – promptly!
  - Run to 5pm,except the last day (1pm)
- Breaks:
  - Drinks at 10:30am and 3pm daily (half hour)
  - Lunch at 12pm daily (one hour)
- Fire alarm test on Wednesday at 1pm

WI-FI Access

- *eduroam* is available as normal around the building
- UoM guest accounts available if you need one
  - Please ask!

Introductory course – Monday 7th

Time	Session	Presenter(s)
12:00	Lunch and Registration	
13:00	Workshop logistics	Simon
13:15	SpiNNaker Hardware & Tools Overview	Simon
14:00	Running PyNN simulations on SpiNNaker	Andrew
15:00	Coffee	
15:30	Lab time	
17:00	Close	

Introductory course – Tuesday 8th

	Session	Owner
09:00	Synaptic plasticity using PyNN	Sergio
10:00	Lab time (with coffee at 10:30)	
12:00	Lunch	
13:00	Simple data I/O and visualisation	Alan
14:00	Lab time (with coffee at 15:00)	
17:00	Close	

Introductory course - Wednesday 9th

Time	Session	Owner
09:00	Free lab time (with coffee at 10:30)	
12:00	Lunch and close	

Advanced Course – Wednesday 9th

Time	Session	Owner
09:00	Registration and coffee	
09:15	SpiNNaker architecture, chip resources and limitations	Steve
10:00	Adding new neuron models	Andrew/Michael
10:30	Coffee	
11:00	Lab time	
12:00	Lunch	
13:00	SpiNNaker system software (SARK)	Steve
13:45	SpiNNaker API	Luis
14:30	Lab time (with coffee at 15:00)	
16:00	Event-driven neural simulation	Alex
16:30	Lab time	
17:00	Close	

Advanced Course – Thursday 10th

Time	Session	Owner
09:00	Maths & fixed point libraries	Michael
09:45	Adding new models of synaptic plasticity	Jaime
10:30	Lab time (with coffee from 10:30)	
12:00	Lunch	
13:00	Connecting to external devices	Alex/Sergio
13:45	Lab time (with coffee at 15:00)	
16:00	Debugging using YBUG & GDB	Steve
17:00	Close	

Advanced Course – Friday 11th

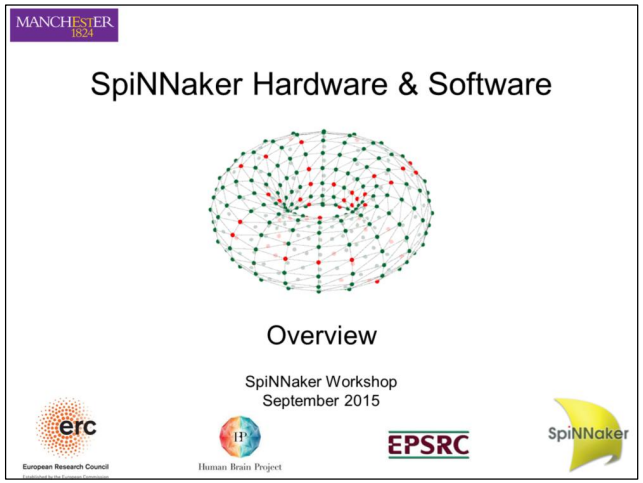
Time	Session	Owner
09:00	Free lab time	
10:30	Coffee	
11:00	Free lab time	
12:00	Lunch and close	

Loan board requests…

- 4-node boards can be loaned out
  - But supply is limited
- Please send an email with your project details to:
  simon.davidson@manchester.ac.uk
- Steve Temple will allocate and log board loans
  - Please don't just take one away….

Feedback…

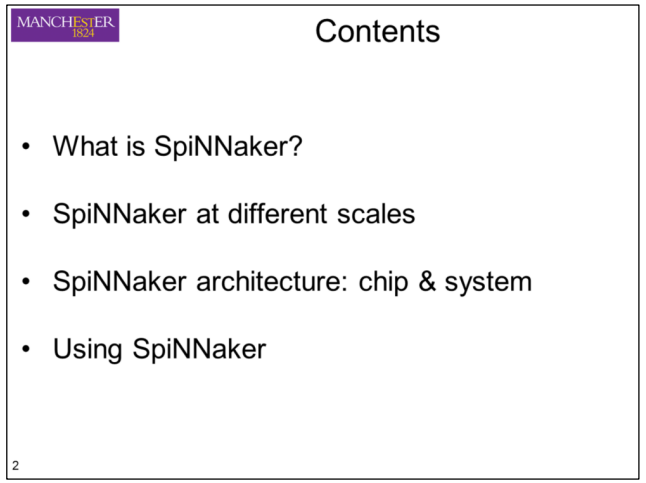
- We'd appreciate some feedback on…
  - Your workshop experience
  - SpiNNaker hardware
  - SpiNNaker software
- I'll email you in the next few weeks
- We hope that you enjoy the workshop!



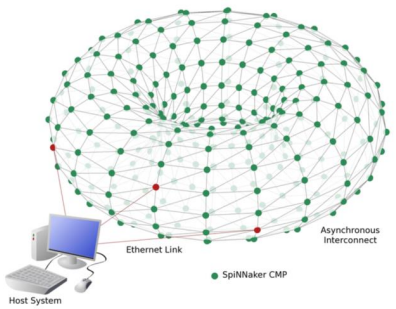
This presentation is to provide a quick overview of the hardware and software of SpiNNaker. It introduces some key concepts of the topology of a SpiNNaker machine, the unique message passing and routing functionality and the chip architecture with its restrictions and limitations.

The other function of this talk is to introduce some terminology that will be used through the workshop.

Much of the information herein will be expanded upon in later talks, so don't be too concerned with remembering everything!



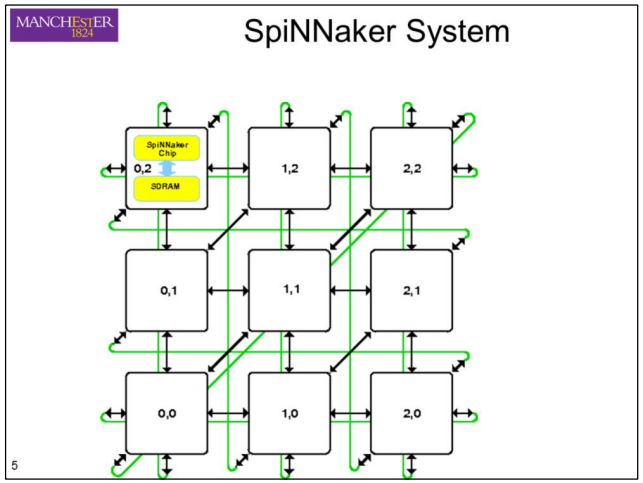
SpiNNaker Project



A million mobile phone processors in one computer
Able to model about 1% of the human brain…
…or 10 mice!

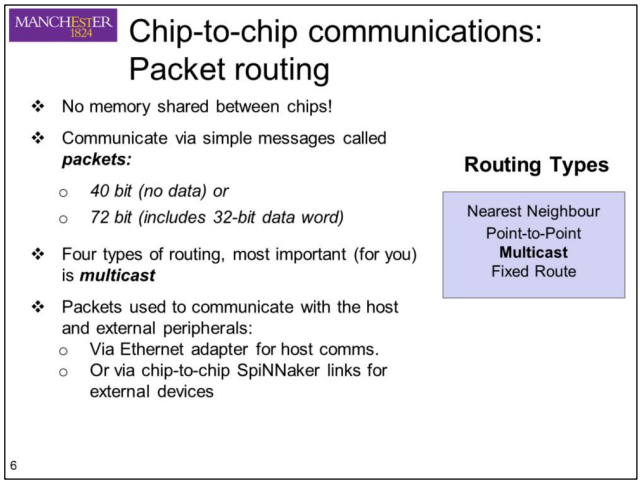
How is SpiNNaker Used?

- Some key user communities:
  - **Computational neuroscientists** to simulate large neural models and try to understand the brain
  - **Roboticists** to build advanced neural sensory and control systems
  - **Computer architects** to apply neural theories of computation to non-neural problems



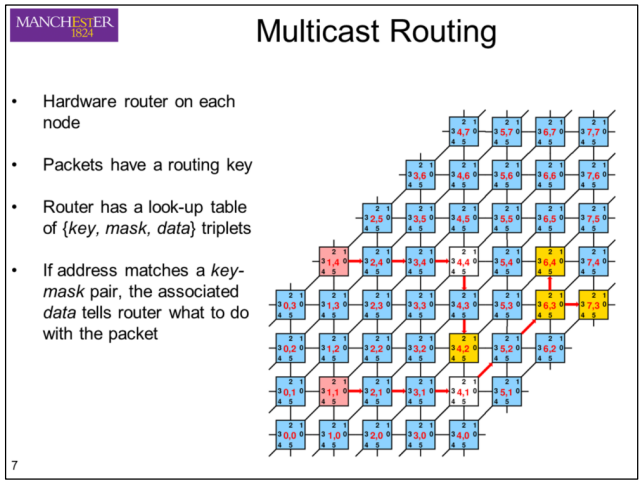
SpiNNaker chips have six links: North, South, East, West, North-East and South-West. Links are bi-directional and work independently.

Topologically, an array of SpiNNaker chips forms a hexagonal grid, which can wrap around to form a cylinder or toroid. Machines can be constructed from an arbitrary sized array of chips, up to 256 x 256 in size.

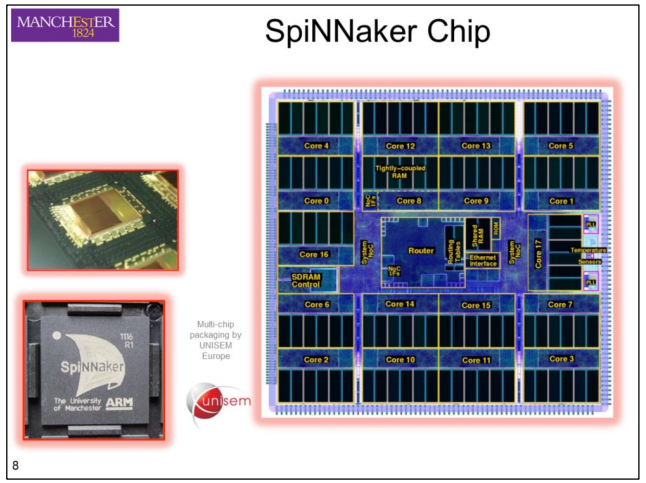


The only way for cores on two different chips to communicate is via simple messages that are passed from one chip to the other until they reach their destination. These messages are called packets. The optional dataword with each packet is called its *payload* and is always 32-bits. When the host reads or writes data to/from SpiNNaker the data is broken down into many of these packets (thought the user does not need to know this!)

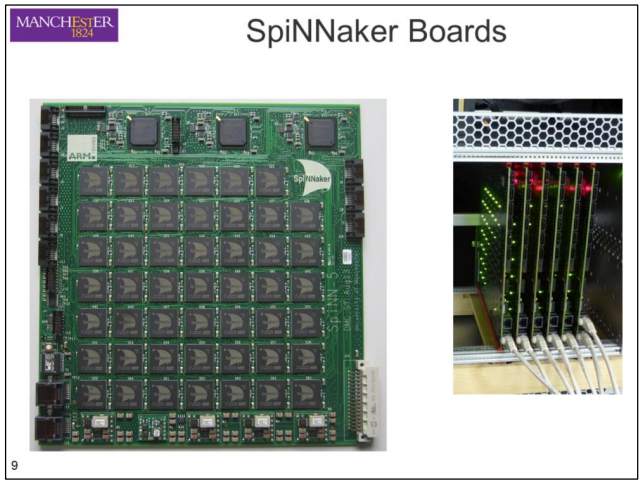
The multicast routing type is the most flexible and is the one most likely to be used in applications. We can provide information about the other routing types if you are interested. They are mainly used for system functions and (in the case of the nearest neighbour type) for flood-filling the machine with common code during the initialisation phase.



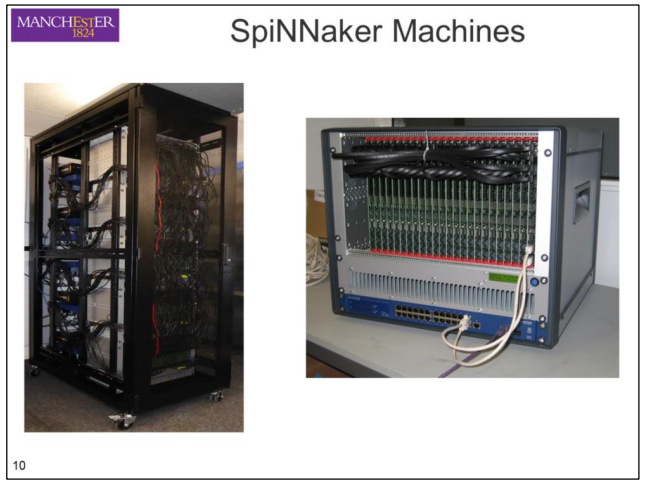
When a packet is generated by a core, it has a 32-bit routing key that identifies its source. The packet is given to the hardware router on its chip which decides what to do with it: to give it to one (or more!) cores on this chip or send it down one (or more!) of the six links to other chips. At each step on its journey the receiving router performs a look-up of the routing key in its 1024 entry table. The first match it finds is a 'hit' and it reads the associated data word in the table to see what action to take. If no match is found, the default behaviour is to send the packet out from the opposite link by which it entered.



The SpiNNaker chip has 18 cores, a hardware router and an interface to 128MB of external SDRAM. All cores are identical. At boot time the first core to complete the boot process becomes the 'monitor core' and the next 16 become 'application' cores. The remaining 18th core may be non-functional as we used chips in which at least 17 cores are working (to improve the yield of useful chips!)

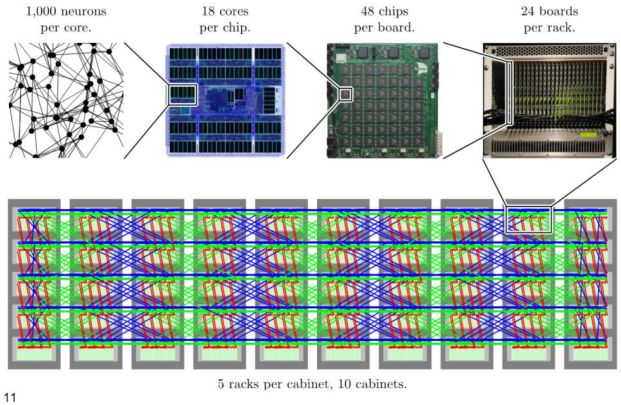


Each SpiNN-5 board (shown on the left) has 48 SpiNNaker chips and three FPGAs, which are used for board-to-board communications. The SpiNNaker links from each chip on the edge go via the FPGAS where they are translated into high-speed serial traffic, sent to the next board via SATA cables and then translated back into SpiNNaker link protocol. This is invisible to the packets themselves. When many bards are put together they form a *subrack* shown on the right.



Each subrack (on the right) can hold up to 24 boards (1152 chips or 20K cores). Five subracks can be stacked to form a cabinet (on the right), containing 100K cores.

Scaling to a billion neurons



11

Connecting ten cabinets together into a single toroid gives us 1 million cores.

Back of the envelope calculations are that a 1 million core machine can simulation between 100 million and 1 billion simple leaky-integrate-and-fire neurons in real time.

What Next for SpiNNaker?

- Five cabinet machine (500K ARM cores)
  - Will be online later this year
  - Will be available for remote access
    - Open to any research project, in principle
- SpiNNaker2 being developed within HBP
  - New systems by 2020?
- For further information contact:
simon.davidson@manchester.ac.uk

12

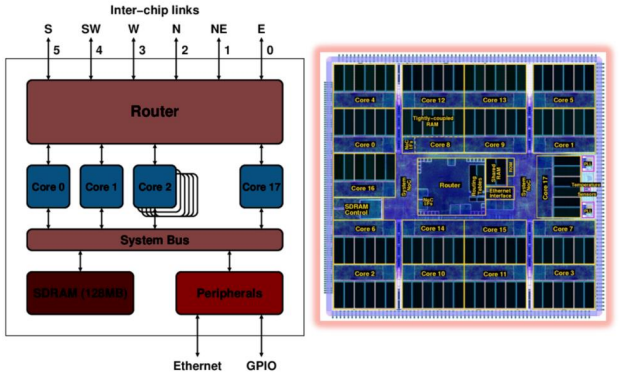
MANCHESTER 1824

Chip Architecture

13

MANCHESTER 1824

SpiNNaker Node



14

As discussed earlier, the 18 cores each share the hardware router (through which they can send packets through the links) and the shared 128MB memory. Each board can connect to a host via an Ethernet adapter.

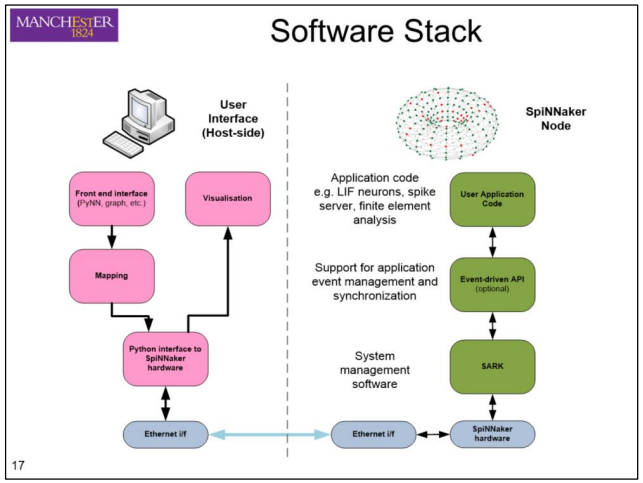
Chip Resources

- 18 cores on a chip:
  - 1 Monitor Processor
  - 16 Application processors
  - 1 fault-tolerant/yield spare
- Each core is an ARM968 processor
  - 200 MHz clock speed
  - No memory management or floating point!
  - Local memories:
    - 32K local code memory (ITCM), 64K local data (DTCM)
    - TCMs are visible only to local processor
- 128MByte SDRAM
  - Shared and visible to all processors on **same node**
- Router:
  - Directs flow of information from core-to-core across the machine

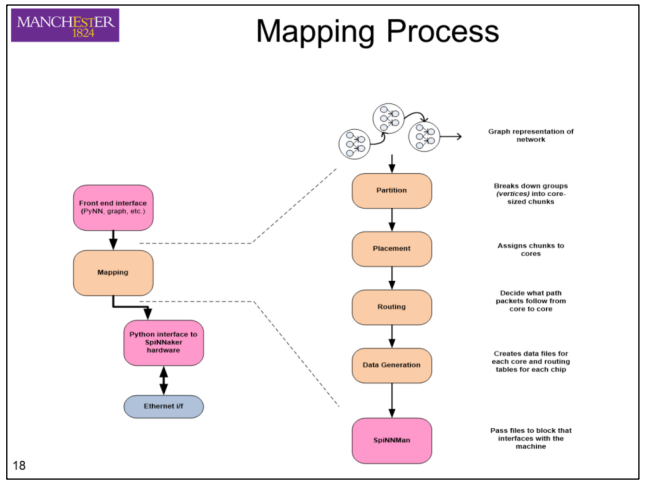
The local 64K data space can be accessed in a single cycle, whereas the shared 128MB memory takes dozens of cycles. Although the typical method of accessing this SDRAM is via DMA, the memory is mapped in the same address space as the DTCM and so it can, in principle be accessed using simple load and store instructions. There is a small, shared on-chip SRAM that has not been mentioned here. It is *mostly* used by the system for housekeeping functions and so is not generally available for applications.

Using SpiNNaker:

The Software Stack



Host-side the software is written in Python (pink boxes). On the machine, software is compile 'c' code (green boxes).
In our software stack, the user species their model in a domain-specific language (such as PyNN) which is translated into a graph like format in which computational elements are vertices and communication between these elements is represented by directed edges. The problem is mapped to the machine (see next slide) and the various files required for each chip and core are generated. This is loaded to SpiNNaker. A SpiNNaker application, running on many cores sits above an (optional) API and a system software layer (SARK). SARK provides essential resource management and comms functions. The API provides a framework for event-driven applications.



This slides zooms in on the mapping part of the host's activities. The user problem is translated into a graph, as described earlier. Each vertex represents some computation (e.g. a group of neurons) and this must be broken down into chunks that can be handled by a single core. This is *partitioning*. Each chunk of work is allocated to one of the cores on the target machine, in the *placement* phase. The edges of the graph, representing communication between these blocks of computation are translated into the routing of messages from one core to another. The output of this *routing* phase is a set of routing tables, one per chip, to be loaded to the machine. In the *data generation* phase the routing tables and any data required for each computation node are written.

What Files are Required for Simulation?

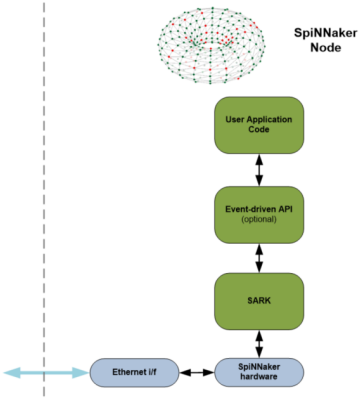
Files Required

For Each Core (16 per chip):

- Application C executable (32KB)
- Application local data (64KB)
- Application shared data (8MB)

For Each Chip:

- Router table
- Any shared data tables



19

Order of Events (batch mode)

1. Compile network description
2. Map graph to machine
3. Generate data files
4. Load files
5. Synchronise the start on all cores!
6. Simulation runs to completion
7. Hands back control to host
8. Read back results and post-process

20

This describes the *batch* mode of operation, typical for running computational neuroscience models such as netorks written in PyNN. If SpiNNaker is used in a robotics environment, it may be set up to running continuously (i.e. without stopping),. It is possible to compile a network once, save all of the files and then re-load them whenever required, which is useful for robotics applications where the network does not change, but the data does.

End of Overview!

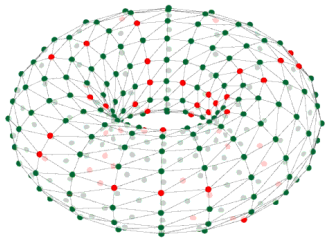
- Much more detail on all of these topics
  - In the sessions to come….
- Any questions for now?
- Just one more thing to add….

Buying SpiNNaker Hardware

- 48-node board now available for sale
- Non-commercial use only
- 4-node boards can only be loaned (currently!)

- For further information contact:
  simon.davidson@manchester.ac.uk

Running PyNN Simulations on SpiNNaker



Andrew Rowley

SpiNNaker Workshop
January 2015

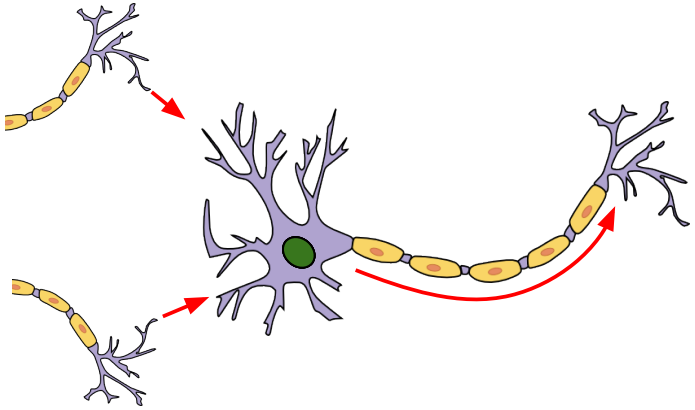
European Research Council
Established by the European Commission

Human Brain Project

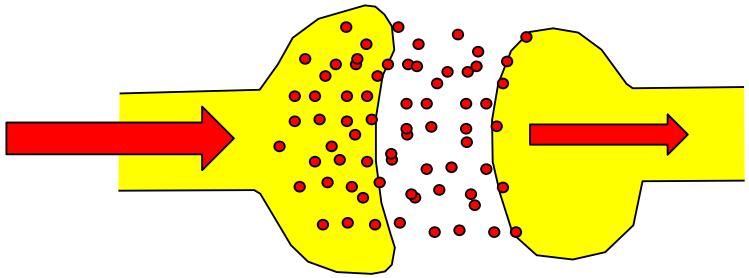
EPSRC

SpiNNaker

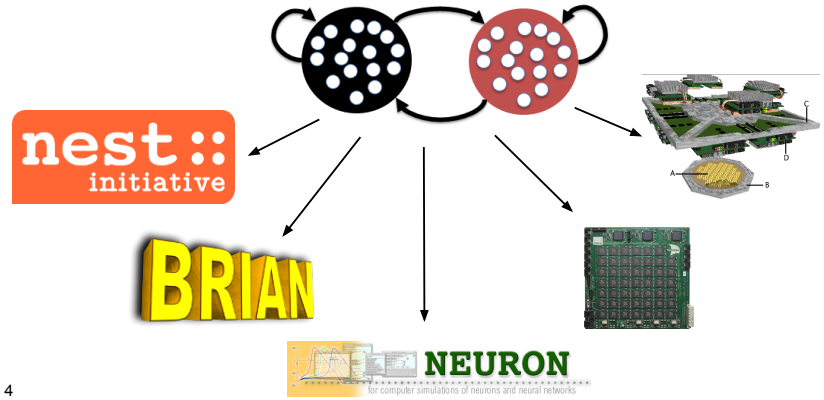
Spiking Neural Networks



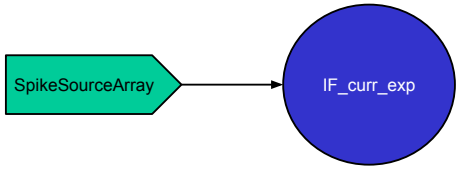
Spiking Neural Networks



What is PyNN?



A Simple PyNN Network



A Simple PyNN Network

```
import pyNN.spiNNaker as p
```


A Simple PyNN Network

```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
```


A Simple PyNN Network

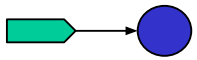
```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
pop_1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
```

A Simple PyNN Network

```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
pop_1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
                     {'spike_times': [0]}, label="input")
```

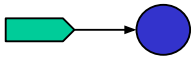
A Simple PyNN Network

```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
pop_1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
                     {'spike_times': [0]}, label="input")
input_proj = p.Projection(input, pop_1, p.OneToOneConnector(
    weights=5.0, delays=1))
```



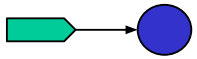
A Simple PyNN Network

```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
pop_1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
                     {'spike_times': [0]}, label="input")
input_proj = p.Projection(input, pop_1, p.OneToOneConnector(
    weights=5.0, delays=1))
pop_1.record()
pop_1.record_v()
```



A Simple PyNN Network

```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
pop_1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
                     {'spike_times': [0]}, label="input")
input_proj = p.Projection(input, pop_1, p.OneToOneConnector(
    weights=5.0, delays=1))
pop_1.record()
pop_1.record_v()
p.run(10)
```



Edit ~/.spynnaker.cfg

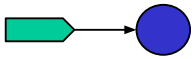
```
[Machine]
#-------
# Information about the target SpiNNaker board or machine:
# machineName:     The name or IP address of the target board
# version:         Version of the Spinnaker Hardware Board (1-5)
# machineTimeStep: Internal time step in simulations in usecs.
# timeScaleFactor: Change this to slow down the simulation time
#                  relative to real time.
#-------
machineName     = None
version         = None
#machineTimeStep = 1000
#timeScaleFactor = 1
```


Edit ~/.spynnaker.cfg

```
[Machine]
#-------
# Information about the target SpiNNaker board or machine:
# machineName:     The name or IP address of the target board
# version:         Version of the Spinnaker Hardware Board (1-5)
# machineTimeStep: Internal time step in simulations in usecs.
# timeScaleFactor: Change this to slow down the simulation time
#                  relative to real time.
#-------
machineName     = 192.168.240.253
version         = 3
#machineTimeStep = 1000
#timeScaleFactor = 1
```

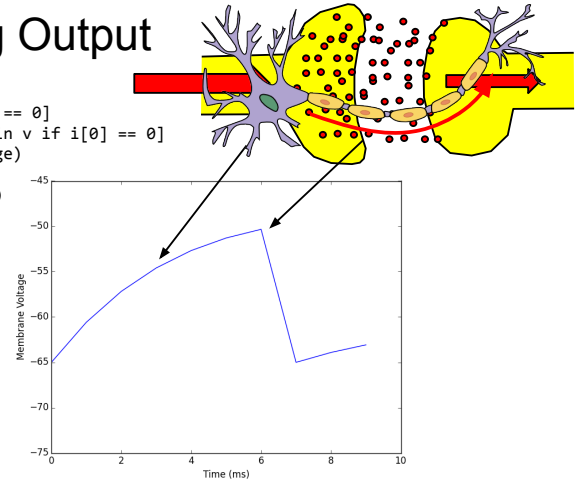

A Simple PyNN Network

```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
pop_1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
                     {'spike_times': [0]}, label="input")
input_proj = p.Projection(input, pop_1, p.OneToOneConnector(
    weights=5.0, delays=1))
pop_1.record()
pop_1.record_v()
p.run(10)
spikes = pop_1.getSpikes()
v = pop_1.get_v()
```



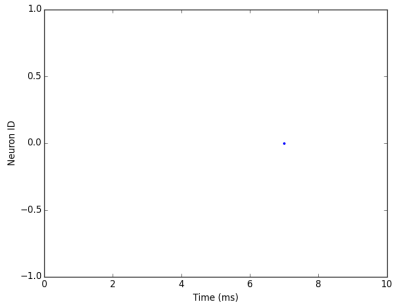
Plotting Output

```
import pylab
time = [i[1] for i in v if i[0] == 0]
membrane_voltage = [i[2] for i in v if i[0] == 0]
pylab.plot(time, membrane_voltage)
pylab.xlabel("Time (ms)")
pylab.ylabel("Membrane Voltage")
pylab.axis([0, 10, -75, -45])
pylab.show()
```

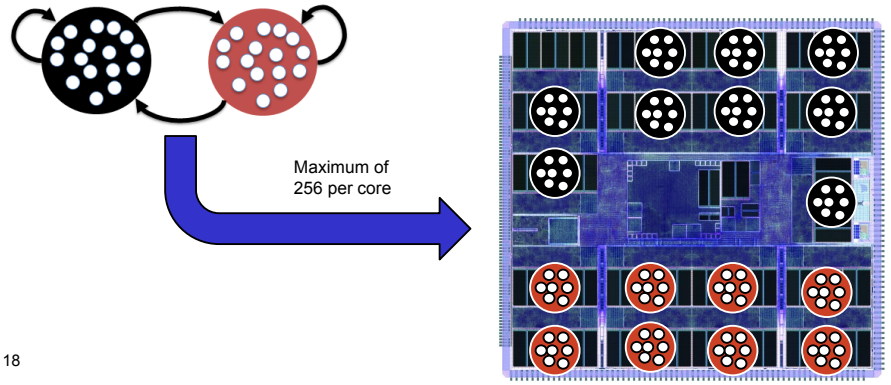


Plotting Output

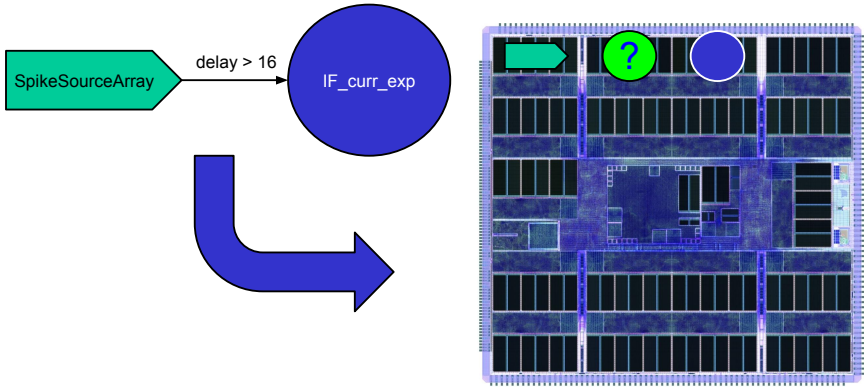
```
import pylab
spike_time = [i[1] for i in spikes]
spike_id = [i[0] for i in spikes]
pylab.plot(spike_time, spike_id, ".")
pylab.xlabel("Time (ms)")
pylab.ylabel("Neuron ID")
pylab.axis([0, 10, -1, 1])
pylab.show()
```



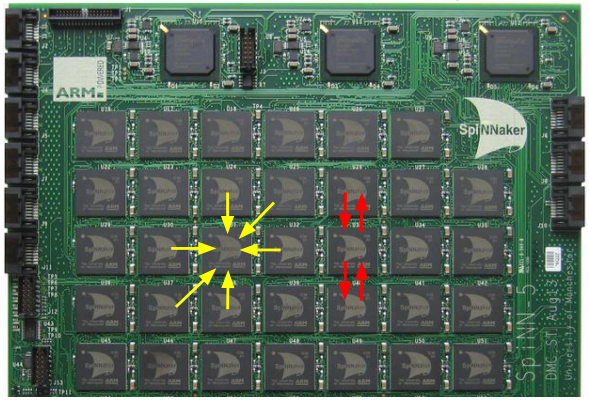
Limitations of PyNN on SpiNNaker:
Neurons Per Core



Limitations of PyNN on SpiNNaker:
Number of cores available

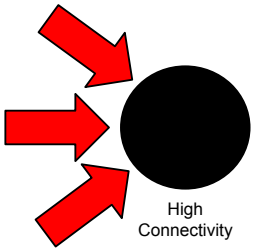


Limitations of PyNN on SpiNNaker:
Dropped Packets (Missing Spikes)

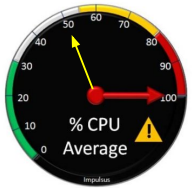


SpiNNaker-Specific PyNN

```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
p.set_number_of_neurons_per_core(p.IF_curr_exp, 100)
pop_1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
```



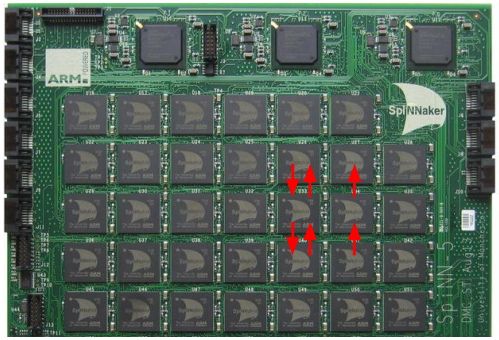
High Connectivity



Plasticity

SpiNNaker-Specific PyNN

```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
pop_1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
pop_1.add_placement_constraint(x=1, y=1)
```



Configuration with spynnaker.cfg

```
[Machine]
machineName      = None
version          = None
timeScaleFactor  = 1
```

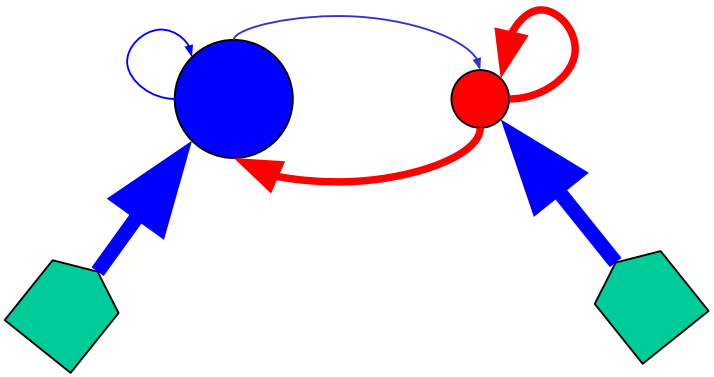
Real Time (ms)	0	10	20	30
Sim Time (ms)	0	10	20	30

Configuration with spynnaker.cfg

```
[Machine]
machineName      = None
version          = None
timeScaleFactor  = 10
```

Real Time (ms)	0	10	20	30
Sim Time (ms)	0	1	2	3

Balanced Random Network

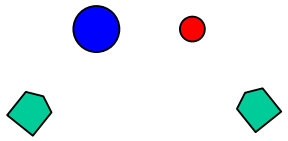


Balanced Random Network

```
import pyNN.spiNNaker as p
import pylab
from pyNN.random import RandomDistribution

p.setup(timestep=0.1)
n_neurons = 1000
n_exc = int(round(n_neurons * 0.8))
n_inh = int(round(n_neurons * 0.2))
```

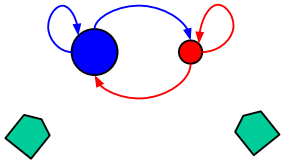

Balanced Random Network

```
pop_exc = p.Population(n_exc, p.IF_curr_exp, {},
                       label="Excitatory")
pop_inh = p.Population(n_inh, p.IF_curr_exp, {},
                       label="Inhibitory")
stim_exc = p.Population(n_exc, p.SpikeSourcePoisson,
                        {"rate": 10.0}, label="Stim_Exc")
stim_inh = p.Population(n_inh, p.SpikeSourcePoisson,
                        {"rate": 10.0}, label="Stim_Inh")
```



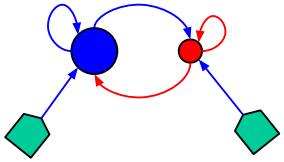
Balanced Random Network

```
conn_exc = p.FixedProbabilityConnector(0.1, weights=0.2,
                                       delays=2.0)
conn_inh = p.FixedProbabilityConnector(0.1, weights=-1.0,
                                       delays=2.0)
p.Projection(pop_exc, pop_exc, conn_exc, target="excitatory")
p.Projection(pop_exc, pop_inh, conn_exc, target="excitatory")
p.Projection(pop_inh, pop_inh, conn_inh, target="inhibitory")
p.Projection(pop_inh, pop_exc, conn_inh, target="inhibitory")
```



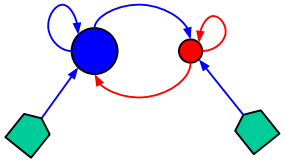
Balanced Random Network

```
delays_stim = RandomDistribution("uniform", [1.0, 1.6])
conn_stim = p.OneToOneConnector(weights=2.0,
                                delays=delays_stim)
p.Projection(stim_exc, pop_exc, conn_stim, target="excitatory")
p.Projection(stim_inh, pop_inh, conn_stim, target="excitatory")
```



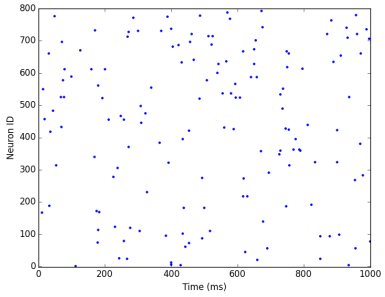
Balanced Random Network

```
pop_exc.initialize("v", RandomDistribution("uniform",
                                           [-65.0, -55.0]))
pop_inh.initialize("v", RandomDistribution("uniform",
                                           [-65.0, -55.0]))
pop_exc.record()
p.run(1000)
```



Balanced Random Network

```
spikes = pop_exc.getSpikes()
pylab.plot([i[1] for i in spikes], [i[0] for i in spikes], ".")
pylab.xlabel("Time (ms)")
pylab.ylabel("Neuron ID")
pylab.axis([0, 1000, -1, n_exc + 1])
pylab.show()
```



5th SpiNNaker Workshop

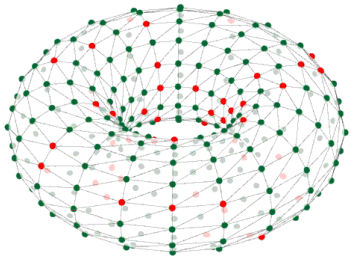
Day 2

September 8th 2015

	Session	Owner
09:00	Synaptic plasticity using PyNN	Sergio
10:00	Lab time (with coffee at 10:30)	
12:00	Lunch	
13:00	Simple data I/O and visualisation	Alan
14:00	Lab time (with coffee at 15:00)	
17:00	Close	

Manchester, UK

Synaptic plasticity on SpiNNaker with PyNN



Sergio Davies

SpiNNaker Workshop
September 2015

European Research Council
Established by the European Commission

Human Brain Project

EPSRC

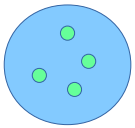
SpiNNaker

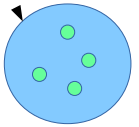
Neural network description

A neural network is usually described in terms of:

- Populations of neurons
- Projections between populations

Each population and projection has its own properties





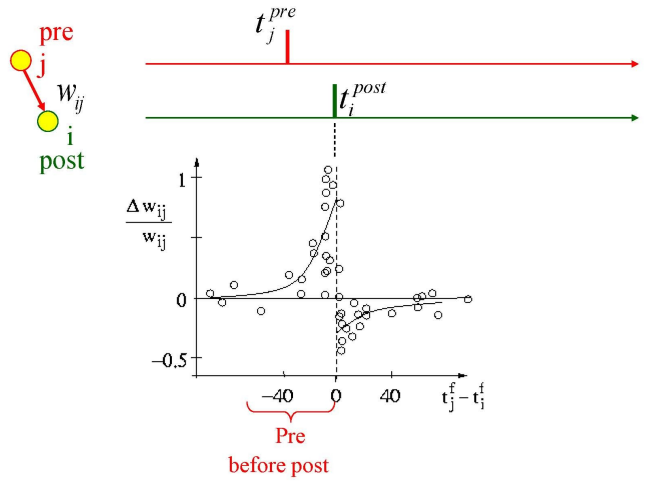
Projections

Attributes of the projections include (but are not limited to):

- Pre-synaptic population
- Post-synaptic population
- Connector type (All-To-All, One-To-One, etc.)
- Target – Synapse type (Excitatory, Inhibitory)
- Static or Dynamic attributes

Plasticity rules (1/2)

STDP spike pair rule



[Sjöström and Gerstner, 2010]

Plasticity rules (2/2)

Other rules available on SpiNNaker in "sPyNNakerExtraModelsPlugin" module:

- Vogels *[Vogels et al. (2011)]*
- Triplet-based rule *[J.-P. Pfister et al. (2006)]*
- *More...*

Behaviour dependence

Behaviour of plasticity rules may depend on one or more parameters:

- Weight dependence
- Time dependence

Weight dependence

- Additive weight dependence
- Multiplicative weight dependence

Weight Dependence example:

```
PyNN.AdditiveWeightDependence
(w_min, w_max, A_plus, A_minus)
```


Timing dependence

- SpikePairRule
- Vogels2011Rule
- Etc.

Timing Dependence example:

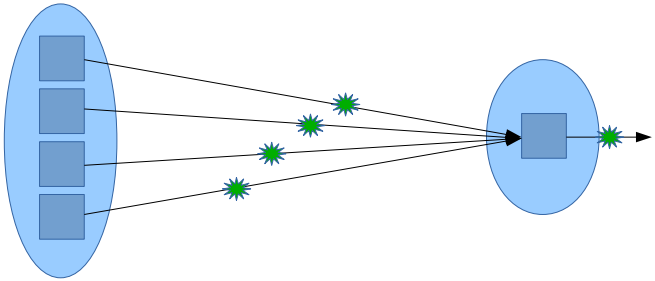
```
PyNN.SpikePairRule
(tau_plus, tau_minus, nearest)
```

Example

- Definition of a learning rule:

```
time_rule = SpikePairRule(tau_plus=1, tau_minus=1)

weight_rule = AdditiveWeightDependence(
        w_min=0.0, w_max=2, A_plus=0.5, A_minus=0.5)

stdp_model = STDPMechanism(
    timing_dependence = time_rule,
    weight_dependence = weight_rule)

syn_dyn = SynapseDynamics(slow = stdp_model)

Projection(pop_src, pop_dst, p.AllToAllConnector(weights,
delays), syn_dyn)
```


Example



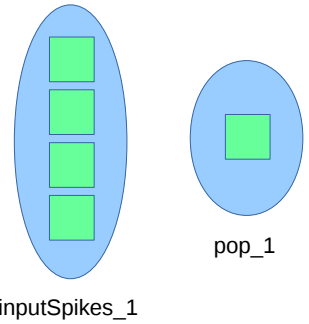
Synapses with long-term plasticity

Building the network – 1

```
import pyNN.spiNNaker as p
import numpy

p.setup(timestep=1.0, min_delay =
1.0, max_delay = 16.0)

end_time = 1100

cell_params_lif = {
    'cm'        : 0.25, # nF
    'i_offset'  : 0.0,
    'tau_m'     : 20.0,
    'tau_refrac': 2.0,
    'tau_syn_E' : 5.0,
    'tau_syn_I' : 5.0,
    'v_reset'   : -70.0,
    'v_rest'    : -65.0,
    'v_thresh'  : -50.0
    }

SpikeArray = {
    'spike_times':
    [range(0,end_time,50),
     range(3,end_time,50),
     range(6,end_time,50),
     range(9,end_time,50)]}

populations = list()
projections = list()
```

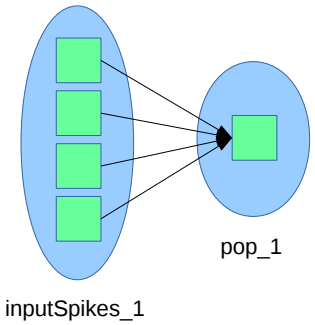

Building the network – 2

```
p.Population(
    4,
    p.SpikeSourceArray,
    spikeArray,
    label='inputSpikes_1')

p.Population(
    1,
    p.IF_curr_exp,
    cell_params_lif,
    label='pop_1')
```



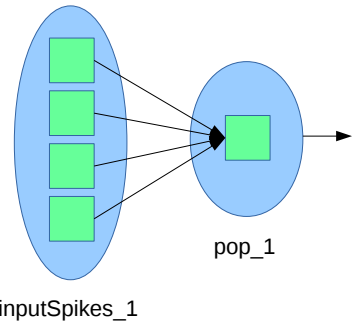
Building the network - 3

```
t_rule = p.SpikePairRule(
    tau_plus=1, tau_minus=1,
    nearest=True)
w_rule = p.AdditiveWeightDependence(
    w_min=0.0,  w_max=2,
    A_plus=0.5, A_minus=0.5)

stdp_model = p.STDPMechanism(
    timing_dependence = t_rule,
    weight_dependence = w_rule,
)

p.Projection(
    populations[0],
    populations[1],
    p.AllToAllConnector(
        weights = weight_to_spike,
        delays = delay),
    synapse_dynamics =
        p.SynapseDynamics(
            slow = stdp_model)))
```

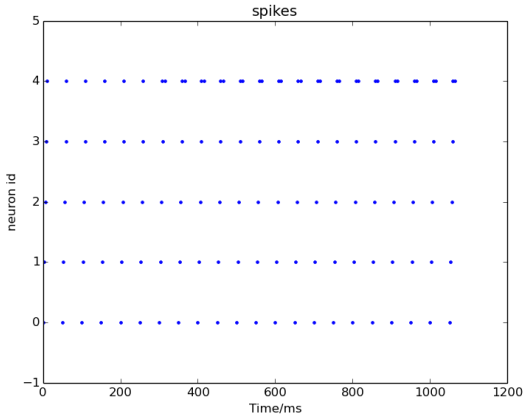


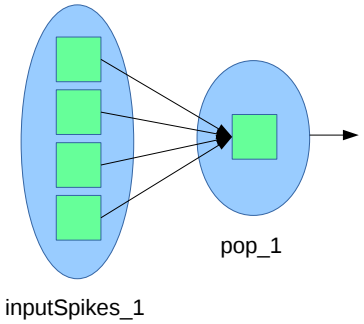
Running the network

```
populations[1].record()

p.run(end_time)

spikes_2 = populations[1].getSpikes()
```



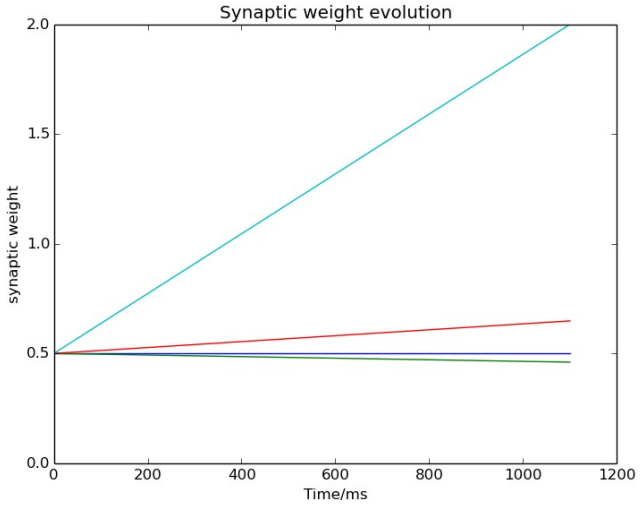
Results – 1



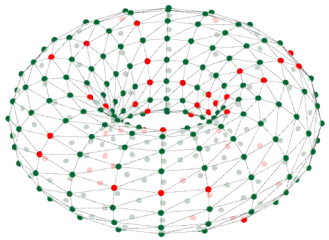


Results – 2

Evolution of the synaptic weights:



Simple Data I/O and visualisation



Alan B Stokes

SpiNNaker Workshop
September 2015

European Research Council
Established by the European Commission

Human Brain Project

EPSRC

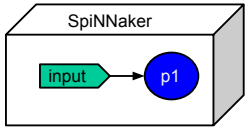
SpiNNaker

Contents

Standard PyNN support (Summary)

- Supports post execution gathering of certain attributes:
  - aka transmitted spikes, voltages etc.

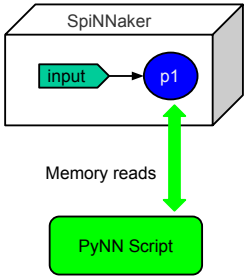
```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
p1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
                {'spike_times': [0]}, label="input")
input_proj = p.Projection(input, p1, p.OneToOneConnector(
  weights=5.0, delays=1))
p1.record()
p1.record_v()
```



Standard PyNN support (Summary)

- Supports post execution gathering of certain attributes:
  - aka transmitted spikes, voltages etc.

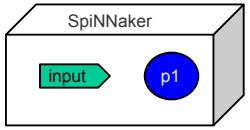
```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
p1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
                {'spike_times': [0]}, label="input")
input_proj = p.Projection(input, p1, p.OneToOneConnector(
  weights=5.0, delays=1))
p1.record()
p1.record_v()
p.run(5000)
spikes = p1.getSpikes()
v = p1.get_v()
```



Standard PyNN support (Summary)

- Supports spike sources of:
  - Spike Source Array, Spike source poisson.

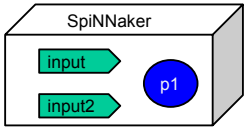
```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
p1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
            {'spike_times': [0]}, label="input")
```



Standard PyNN support (Summary)

- Supports spike sources of:
  - Spike Source Array, Spike source poisson.

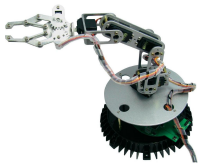
```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
p1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
            {'spike_times': [0]}, label="input")
input2 = p.Population(1, p.SpikeSourcePoisson,
            {'rate':100, 'duration':50}, label='input2')
```



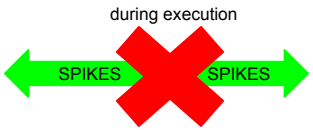
Standard PyNN support (Summary)

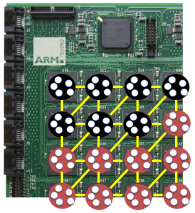
Restrictions

1. Recorded data is stored on SDRAM on each chip.
2. Data to be injected has to be known up-front, or rate based.
3. No support for closed loop execution with external devices.



http://www.conrad.com/ce/en/product/191516/Arexx-RA1-PRO-Metallic-Robot-Arm



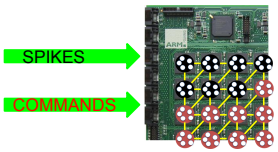


External Device Plugin

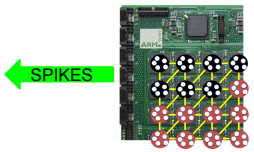
Why? what?

1. Contains functionality for PyNN scripts.
2. Not official PyNN!!!

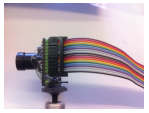
What does it Includes?



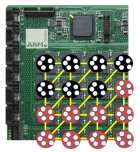
1. Live injection of events and commands into SpiNNaker



2. Live streaming of events from populations.



SPIKES via spinnLink interface



3. External devices support: Covered on Thursday at 13:00

External Device Plugin

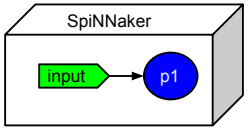
Caveats:

- Injection and live output currently only usable only with the ethernet connection,
- Limited bandwidth of:
  - A small number of spikes per millisecond time step, per ethernet,
  - Shared with both injection and live output,
- Best effort communication,
- Has a built in latency,
- Spinnaker commands not supported by other simulators,
- Loss of cores for injection and live output support,
- You can only feed a live population to one place.

Injecting spikes into PyNN scripts

PyNN script changes

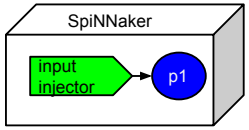
```
import pyNN.spiNNaker as p

p.setup(timestep=1.0)
p1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
                     {'spike_times': [0]}, label="input")
input_proj = p.Projection(input, p1, p.OneToOneConnector(
  weights=5.0, delays=1))
# loop(synfire connection)
loop_forward = list()
for i in range(0, n_neurons - 1):
        loop_forward.append((i, (i + 1) % n_neurons, weight_to_spike, 3))
Frontend.Projection(pop_forward, pop_forward, Frontend.FromListConnector(loop_forward))
```



Injecting spikes into PyNN scripts

PyNN script changes: Declaring an injector population

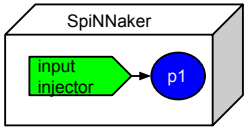
```
import pyNN.spiNNaker as p
import spynnaker_external_devices_plugin.pyNN as ExternalDevices
p.setup(timestep=1.0)
p1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input_injector = p.Population(1, ExternalDevices.SpikeInjector,
                     {'port':95768}, label="injector)
input_proj = p.Projection(input_injector, p1, p.OneToOneConnector(
  weights=5.0, delays=1))
# loop(synfire connection)
loop_forward = list()
for i in range(0, n_neurons - 1):
        loop_forward.append((i, (i + 1) % n_neurons, weight_to_spike, 3))
Frontend.Projection(pop_forward, pop_forward, Frontend.FromListConnector(loop_forward))
```



Injecting spikes into PyNN scripts

PyNN script changes: Setting up python injector

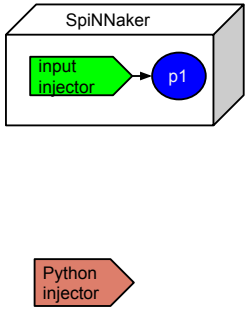
```
.....
# create python injector
def send_spike(label, sender):
        sender.send_spike(label, 0, send_full_keys=True)
```



Injecting spikes into PyNN scripts

PyNN script changes: Setting up python injector

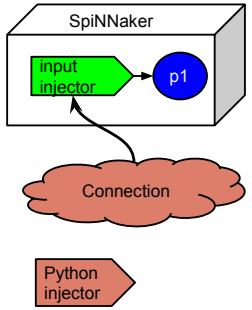
```
.....
# create python injector
def send_spike(label, sender):
        sender.send_spike(label, 0, send_full_keys=True)
# import python injector connection
from spynnaker_external_devices_plugin.pyNN.connections.\
spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection
```



Injecting spikes into PyNN scripts

PyNN script changes: Setting up python injector

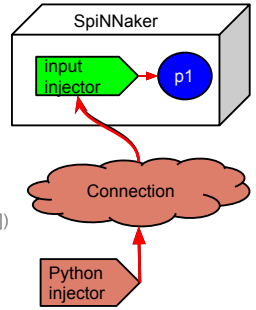
```
.....
# create python injector
def send_spike(label, sender):
        sender.send_spike(label, 0, send_full_keys=True)
# import python injector connection
from spynnaker_external_devices_plugin.pyNN.connections.\
spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection
# set up python injector connection
live_spikes_connection = SpynnakerLiveSpikesConnection(
   receive_labels=None, local_port=19996, send_labels=["spike_sender"])
```



Injecting spikes into PyNN scripts

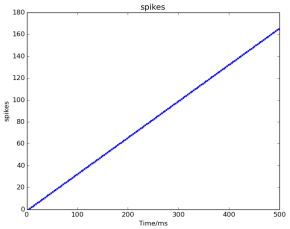
PyNN script changes: Setting up python injector

```
.....
# create python injector
def send_spike(label, sender):
        sender.send_spike(label, 0, send_full_keys=True)
# import python injector connection
from spynnaker_external_devices_plugin.pyNN.connections.\
spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection
# set up python injector connection
live_spikes_connection = SpynnakerLiveSpikesConnection(
        receive_labels=None, local_port=19996, send_labels=["spike_sender"])
# register python injector with injector connection
live_spikes_connection.add_start_callback("spike_sender",  send_spike)
p.run(500)
```



Injecting spikes into PyNN scripts

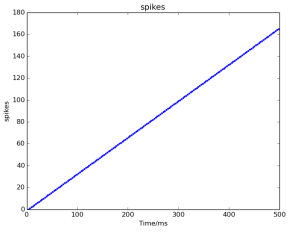
Behaviour with (SpikeSourceArray)



Behaviour with Live injection!

SAME!!!!!

BUT BORING!!!!



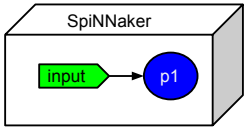
DEMO TIME!!! Injection

DEMO!!!!

Live output from PyNN scripts

PyNN script changes: declaring live output population

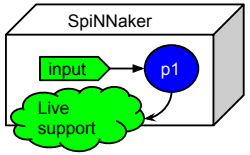
```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
p1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
            {'spike_times': [0]}, label="input")
input_proj = p.Projection(input, p1, p.OneToOneConnector(
  weights=5.0, delays=1))
```



Live output from PyNN scripts

PyNN script changes: declaring live output population

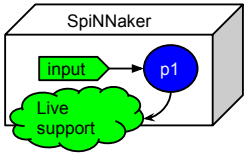
```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
p1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
            {'spike_times': [0]}, label="input")
input_proj = p.Projection(input, p1, p.OneToOneConnector(
  weights=5.0, delays=1))
# declare a live output for a given population.
import spynnaker_external_devices_plugin.pyNN as ExternalDevices
ExternalDevices.activate_live_output_for(p1)
```



Live output from PyNN scripts

PyNN script changes: python receiver

```
.....
# declare python code when received spikes for a timer tick
def receive_spikes(label, time, neuron_ids):
    for neuron_id in neuron_ids:
        print "Received spike at time {} from {}-{}"
            .format(time, label, neuron_id)
```

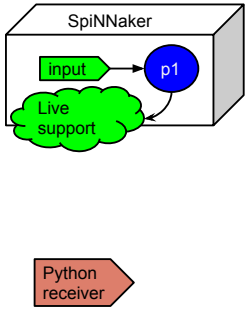


Python receiver

Live output from PyNN scripts

PyNN script changes: python receiver

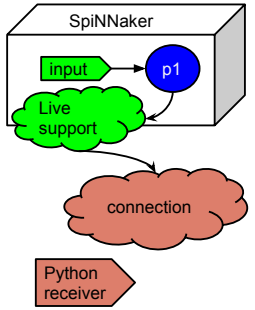
```
.....
# declare python code when received spikes for a timer tick
def receive_spikes(label, time, neuron_ids):
        for neuron_id in neuron_ids:
                print “Received spike at time {} from {}-{}”
                        .format(time, label, neuron_id)
# import python live spike connection
from spynnaker_external_devices_plugin.pyNN.connections.\
spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection
```



Live output from PyNN scripts

PyNN script changes: python receiver

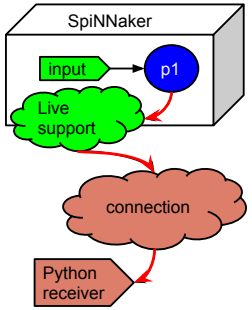
```
.....
# declare python code when received spikes for a timer tick
def receive_spikes(label, time, neuron_ids):
        for neuron_id in neuron_ids:
                print “Received spike at time {} from {}-{}”
                        .format(time, label, neuron_id)
# import python live spike connection
from spynnaker_external_devices_plugin.pyNN.connections.\
spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection
# set up python live spike connection
live_spikes_connection = SpynnakerLiveSpikesConnection(
   receive_labels=[“receiver”], local_port=19995, send_labels=None)
```



Live output from PyNN scripts

PyNN script changes: python receiver

```
.....
# declare python code when received spikes for a timer tick
def receive_spikes(label, time, neuron_ids):
        for neuron_id in neuron_ids:
                print “Received spike at time {} from {}-{}”
                        .format(time, label, neuron_id)
# import python live spike connection
from spynnaker_external_devices_plugin.pyNN.connections.\
spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection
# set up python live spike connection
live_spikes_connection = SpynnakerLiveSpikesConnection(
        receive_labels=[“receiver”], local_port=19995, send_labels=None)
# register python receiver with live spike connection
live_spikes_connection.add_receive_callback(“receiver”, receive_spikes)
p.run(500)
```



DEMO TIME!!! receive live spikes

DEMO!!!!

Visualisation

How current supported visualisations work:

1. Uses the live output functionality as discussed previously.
2. Uses the c based receiver and is planned to be open source for users to augment with their own special visuals.
3. Currently contains raster plot support.

Visualisation

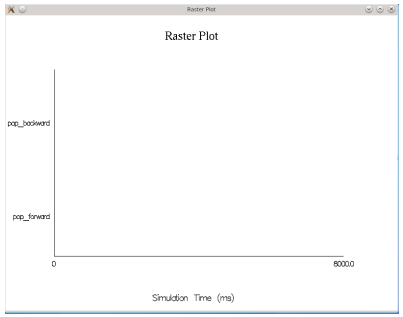
```
cspc277-visualiser-) make -f Makefile.linux
cspc277-visualiser-) .............
cspc277-visualiser-) ./vis -colour_map test_data/spikeio_colours
cspc277-visualiser-)
awaiting tool chain hand shake to say database is ready
```

Input parameters:

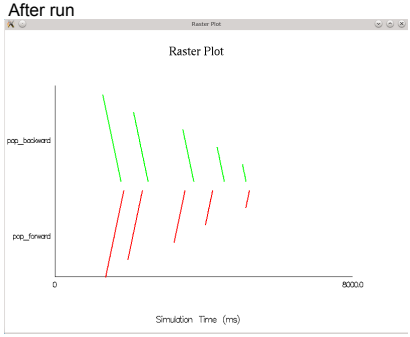
- -colour_map
  - Path to a file containing the population labels to receive, and their associated colours
- -hand_shake_port
  - optional port which the visualiser will listen to for database hand shaking
- -database
  - optional file path to where the database is located, if needed for manual configuration
- -remote_host
  - optional remote host, which will allow port triggering

Visualisation

Before run



After run



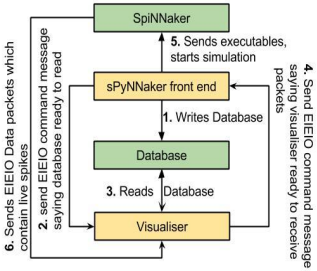
DEMO TIME!!! visualiser and injection of spikes

DEMO!!!!

Technical Detail!!!

Notification protocol under the hood!

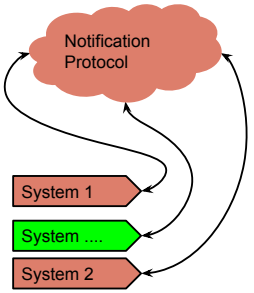
- Everything so far uses the notification protocol.
- It supplies data to translate spikes into population ids.
- If you have more than 1 system running to inject and/or receive, then you need to register this with the notification protocol.



Injecting spikes into PyNN scripts

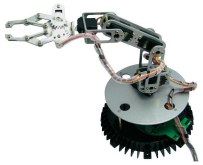
PyNN script changes:
registering a system to the notification protocol

```
.....
# register socket addresses for each system
p.register_database_notification_request(
    hostname="local_host"
    notify_port=19990,
    ack_port=19992)
p.register_database_notification_request(
    hostname="local_host"
    notify_port=19993,
    ack_port=19987)
p.register_database_notification_request(
    hostname="local_host"
    notify_port=19760,
    ack_port=19232)
```



Thanks for listening

Any questions?!



5th SpiNNaker Workshop

Day 3

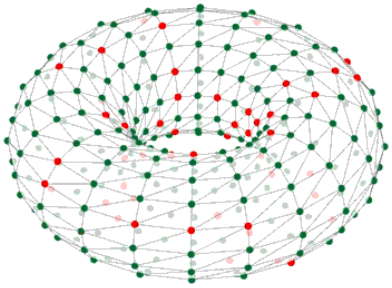
September 9th 2015

Time	Session	Owner
09:00	Free lab time (with coffee at 10:30)	
12:00	Lunch and close	

Time	Session	Owner
09:00	Registration and coffee	
09:15	SpiNNaker architecture, chip resources and limitations	Steve
10:00	Adding new neuron models	Andrew/Michael
10:30	Coffee	
11:00	Lab time	
12:00	Lunch	
13:00	SpiNNaker system software (SARK)	Steve
13:45	SpiNNaker API	Luis
14:30	Lab time (with coffee at 15:00)	
16:00	Event-driven neural simulation	Alex
16:30	Lab time	
17:00	Close	

Manchester, UK

SpiNNaker Chip Resources



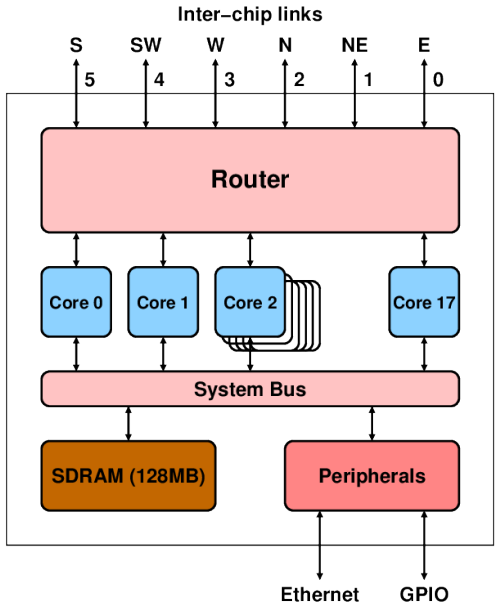
Steve Temple
SpiNNaker Workshop – Manchester – Sep 2015

Overview

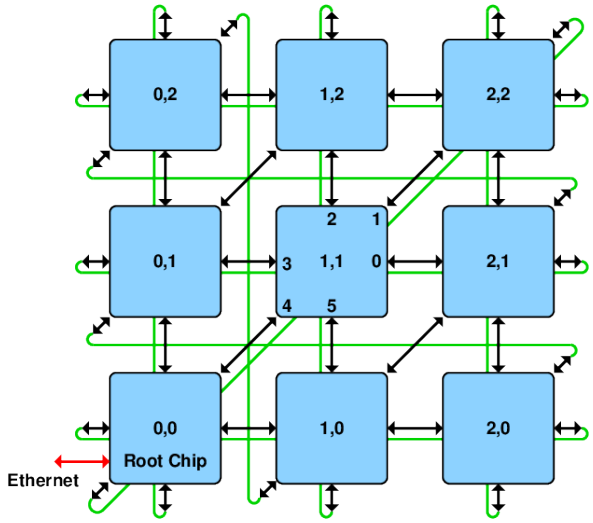
- Chip Architecture
- Core Architecture
- Low-level Communication
  - Packet formats
  - Multicast routing
- High-level Communication – SDP
- Hardware Limitations

Please interrupt if you have a question!

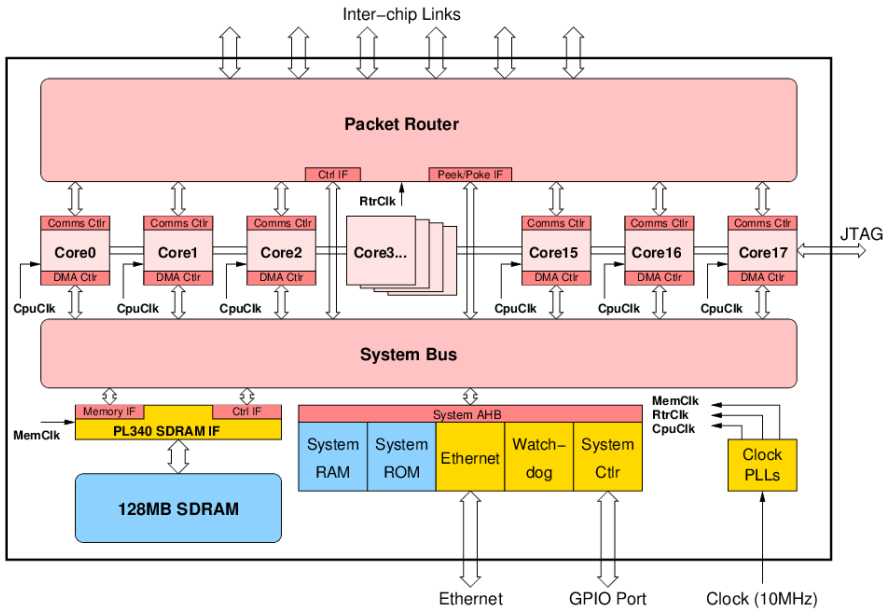
SpiNNaker Chip Outline



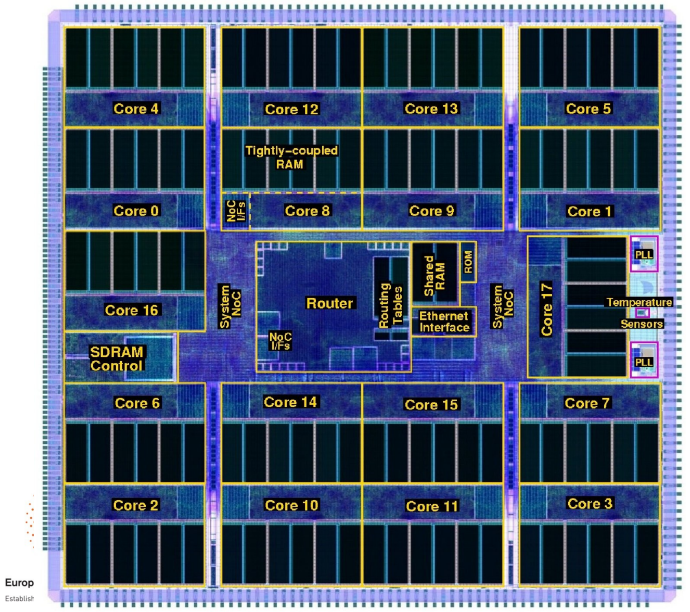
Chip Interconnect



SpiNNaker Chip Details

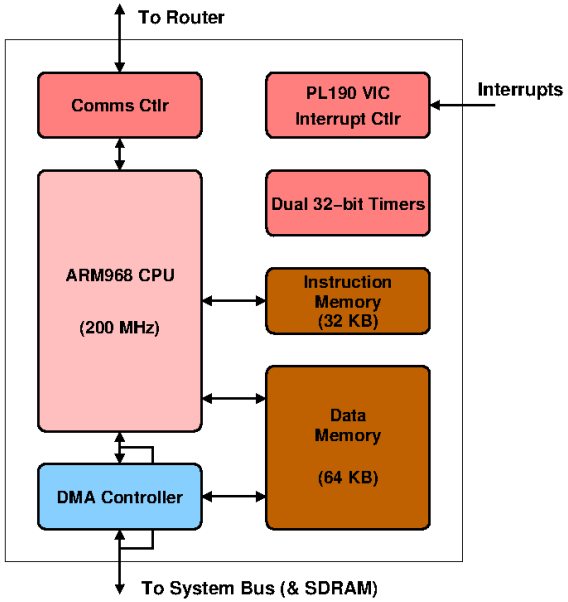


SpiNNaker Chip Layout



- 130nm process
- 10 x 10 mm
- 18 ARM cores with 96K SRAM
- Router
- SDRAM controller
- Asynchronous NoC

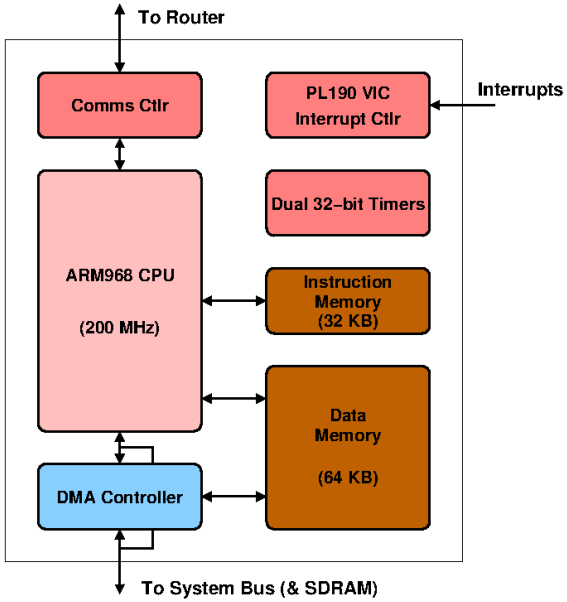
SpiNNaker Core



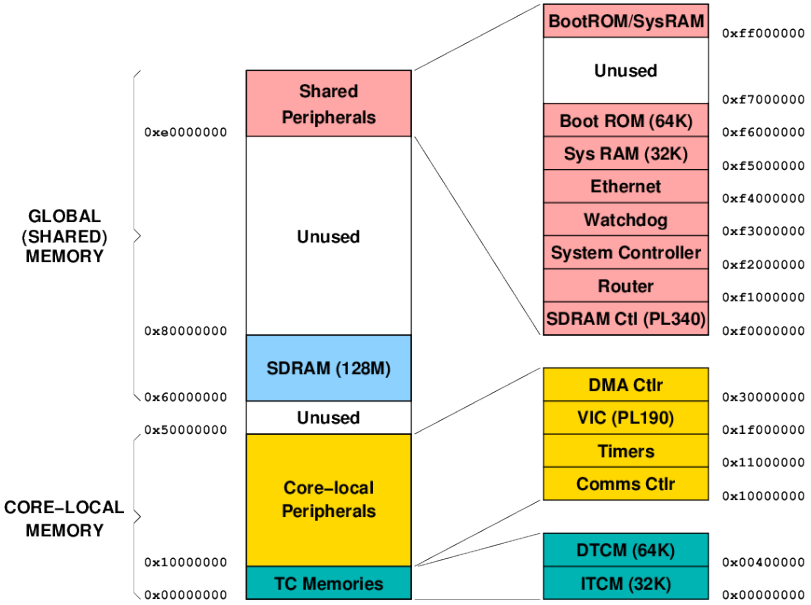
ARM968 CPU

- ARM9 CPU clocked at 200 MHz
- ARM v5TE architecture
  - Supports 32-bit ARM and 16-bit Thumb code
  - Some DSP instruction support - saturated arithmetic, extended multiplies
  - **No floating point hardware!**
- Two Tightly Coupled Memory (TCM) blocks
  - Single cycle (5 ns) access time
  - 32 KB Instruction TCM (ITCM)
  - 64 KB Data TCM (DTCM)
- DMA interface into both TCMs

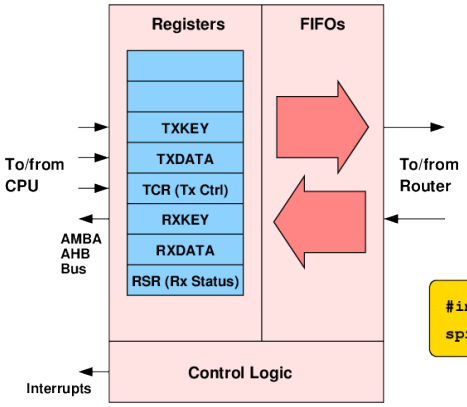
SpiNNaker Core



SpiNNaker Memory Map



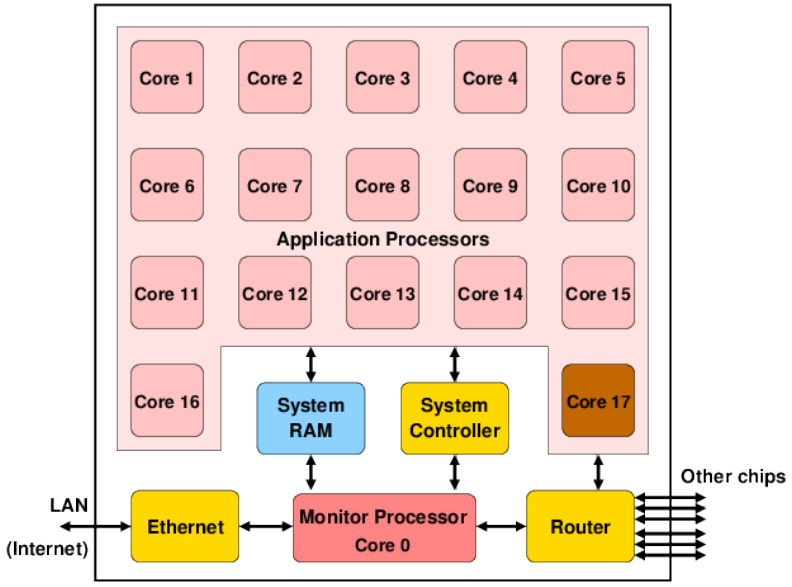
Communications Controller



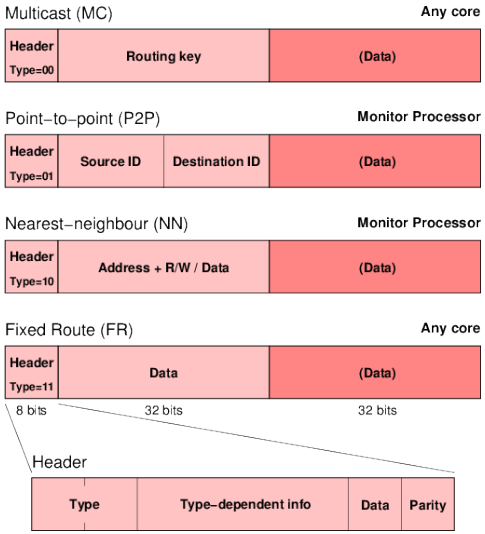

```
#include "spinnaker.h"

void send_mc_pkt (uint key, uint data)
{
  while (cc[TCR] & TX_FULL)
    continue;
  cc[TCR] = PKT_MC_PL;
  cc[TXDATA] = data;
  cc[TXKEY] = key;
}
```

```
#include "spin1_api.h"
spin1_send_mc_pkt (key, data, WITH_PAYLOAD);
```

Monitor Processor & Virtual Cores

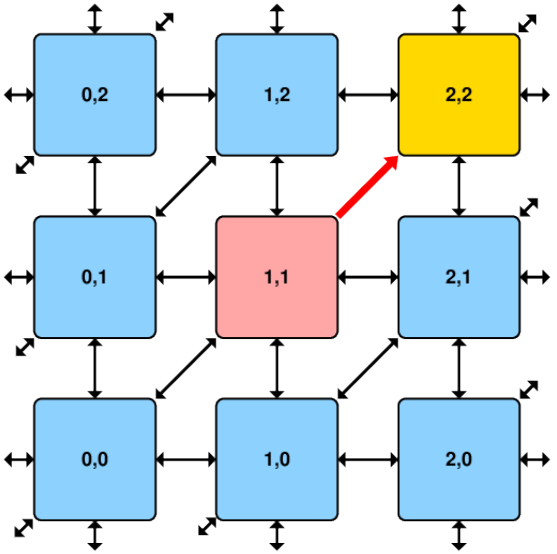


SpiNNaker Packet Types

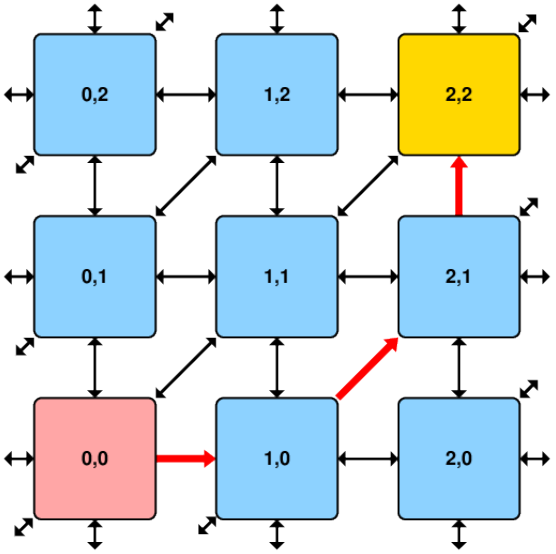



```
uint spin1_send_mc_pkt (uint key, uint data, uint payload);
```

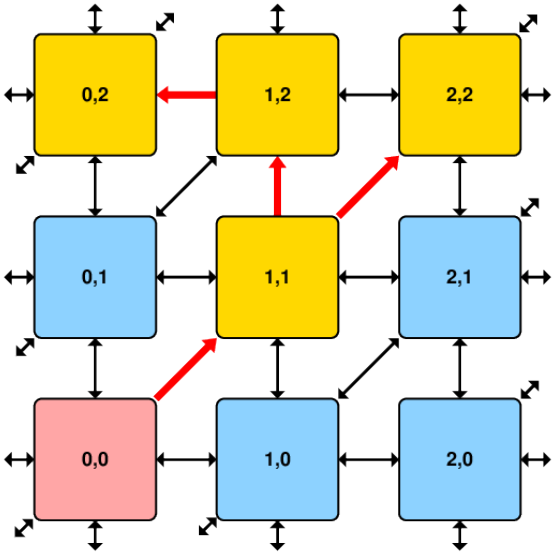
Nearest-neighbour packets



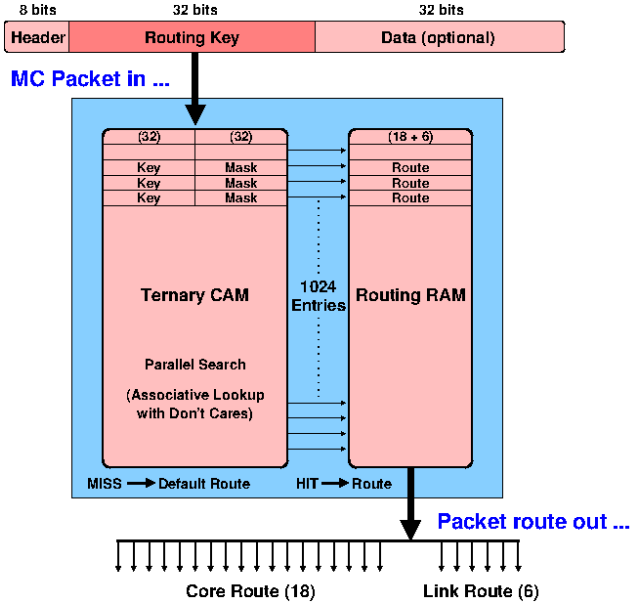
Point-to-point packets



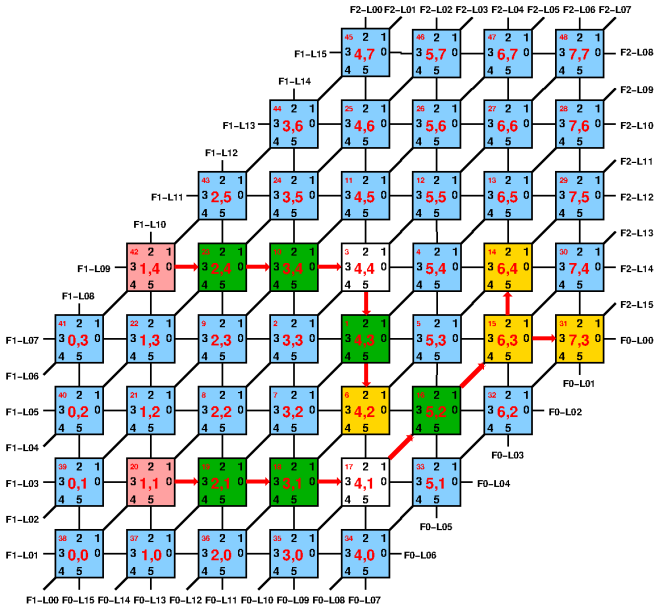
Multicast packets



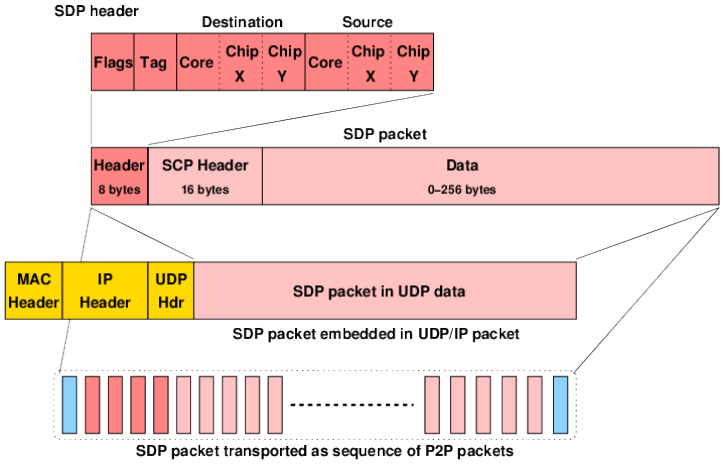
Multicast Packet Router



Multicast Packet Routing

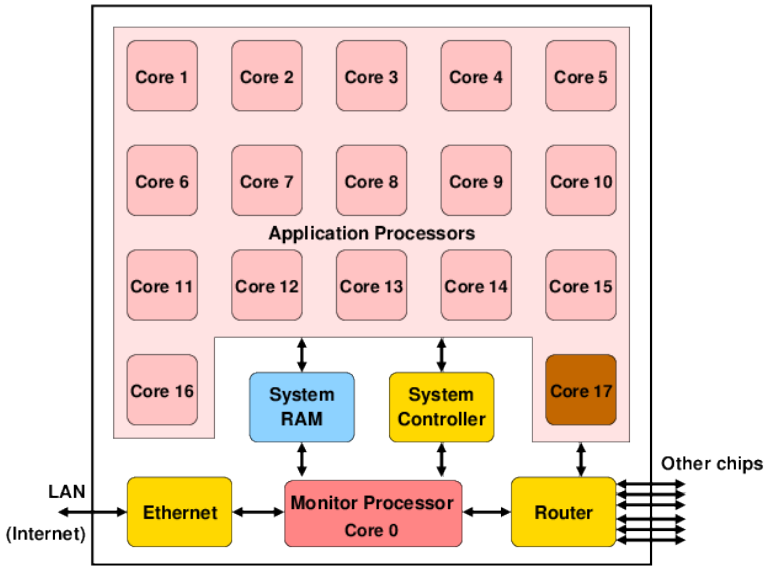


SpiNNaker Datagram Protocol




```
uint spin1_send_sdp_msg (sdp_msg_t *msg, uint timeout);
```

SDP Routing



SpiNNaker Hardware Limits

- Processors – 16/17 per chip (but scalable to thousands of chips)
- ARM968 – ARM9 at 200MHz – 220 DMIPS
- Local memory – very limited
  - Instruction memory – 32K bytes
  - Data memory – 64K bytes
- Local Memory access time - 5 ns
- Per chip memory – 128M bytes (shared)
- Shared memory access time
  - Individual accesses - > 100 ns (NB write buffer)
  - DMA accesses ~ 15ns per word

SpiNNaker Arithmetic Limits

- ARM968 has no floating point hardware
- Options
  - Soft Floating Point – slow and memory hungry
  - Fixed point – uses integer ops
    - Limited range before precision lost
    - Some GCC compiler support (but slowish)
    - Or hand code (C or assembly) for best performance (some libraries available)
- ARM968 has some DSP extensions
  - Saturation, MAC, double operations, CLZ
  - Accessible via compiler intrinsics

SpiNNaker Packet Limits

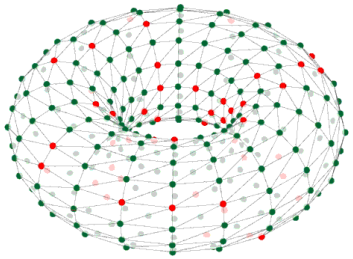
- Packet payload is small – typically 32 bits
- Packet bandwidth is limited
- Chip-to-chip links ~ 250M bit/s (5 or 3 M pkt/s)
  - Currently 50% slower via board-to-board links
- CPU packet processing overhead typically 200-1000ns
- Packets can get lost (dropped) in case of congestion – can be “re-injected” in some cases
- Multicast router table is not infinite!

SpiNNaker Bandwidth Limits

- Overall I/O bandwidth into the machine is limited
- Currently most external I/O is by 100 Mbit/s Ethernet (and only one interface per board)
- High level I/O via SDP is limited by software overheads
  - Around 10 Mbyte/s to Ethernet-attached chip
  - Around 2 Mbyte/s to 'unattached' chips (via P2P packets)
- Potential for higher I/O bandwidth via SATA links on FPGAs but currently unexploited

That's all for now – any questions?

Adding New Neuron Models



Michael Hopkins and Andrew Rowley

SpiNNaker Workshop
September 2015

European Research Council
Established by the European Commission

Human Brain Project

EPSRC

SpiNNaker

Required code separation

- Any new neuron model requires both C and Python code
- C code makes the actual executable (on SpiNNaker), Python code configures the setup and load phases (on the host)
- These are separate but must be perfectly coordinated
- In almost all cases, the C code will be solving an ODE which describes how the neuron state evolves over time and in response to input

We will first describe the C requirements...

Data Structures and Parameters

- The parameters and state of your neuron at any point in time need to be stored in memory
- For each neuron, the C header defines the ordering and size of each stored value
- The C types can be standard integer and floating-point, or ISO draft standard fixed-point, as required (see later talk *Maths & fixed-point libraries*)
- There is also one global data structure which services all neurons on a core

So here is an example using the Izhikevich neuron...

Specific neuron model – data structure

```
#include <neuron-model.h>

//  Izhikevic neuron data structure

typedef struct neuron_t {

// 'fixed' parameters - abstract units
   REAL    A;
   REAL    B;
   REAL    C;
   REAL    D;

// variable-state parameters
   REAL    V;          // nominally in [mV]
   REAL    U;

// offset current [nA]
   REAL    I_offset;

// private variable used internally in C code
   REAL    this_h;

} neuron_t;

...
```

Global data structure

```
...

/*
    Global data structure defined in neuron_model_izh_curr_impl.h
*/

typedef struct global_neuron_params_t {

// Machine time step in milliseconds
   REAL    machine_timestep_ms;

} global_neuron_params_t;
```

Implementing the state update

- Neuron models are typically described as systems of initial value ODEs
- At each time step, the internal state of each neuron needs to be updated in response to inherent dynamics and synaptic input
- There are many ways to achieve this; there will usually be a 'best approach' (in terms of balance between accuracy & efficiency) for each neuron model
- An upcoming paper gives a lot more detail: Hopkins & Furber (2015), “Accuracy and Efficiency in Fixed-Point Neural ODE Solvers” , *Neural Computation* **27**, 1–35.
- The key function will always be *neuron_model_state_update*(); the other functions are mainly to support this and allow debugging etc.

  Continuing the example by describing the key interfaces...

Neuron model API

```
// pointer to a neuron data type - used in all access operations
typedef struct neuron_t* neuron_pointer_t;


// set the global neuron parameters
void neuron_model_set_global_neuron_params(
    global_neuron_params_pointer_t params);


// converts raw value from ring buffer into correctly scaled input
static input_t neuron_model_convert_input(input_t input);


// key function in timer loop that updates neuron state vars and returns spike
// state
bool neuron_model_state_update(input_t exc_input, input_t inh_input,
                               input_t external_bias, neuron_pointer_t neuron);


// return membrane voltage (= first state variable) for a given neuron
state_t neuron_model_get_membrane_voltage(restrict neuron_pointer_t neuron);


// print out neuron definition and state variables
void neuron_model_print(restrict neuron_pointer_t neuron);
```

Specific neuron model – key functions

```
/* simplified version of Izhikevic neuron code */

// make the discrete changes to state after a spike has occurred
static inline void _neuron_discrete_changes(neuron_pointer_t neuron) {

    neuron->V  = neuron->C;     // reset membrane voltage
    neuron->U += neuron->D;     // offset 2nd state variable
}

// key function in timer loop that updates neuron state vars and returns spike state
bool neuron_model_state_update(input_t exc_input, input_t inh_input,
                               input_t external_bias, neuron_pointer_t neuron) {
// collect inputs
    input_this_timestep = exc_input - inh_input + external_bias + neuron->I_offset;

// most balanced ESR update found so far
    _rk2_kernel_midpoint( neuron->this_h, neuron, input_this_timestep );

// test for threshold crossing
    bool spike = REAL_COMPARE( neuron->V, >=, V_threshold );

    If ( spike ) {
       _neuron_discrete_changes( neuron );

// simple threshold correction - next timestep (only) gets a bump
        neuron->this_h = global_params->machine_timestep_ms * SIMPLE_TQ_OFFSET;
    } else {
       neuron->this_h = global_params->machine_timestep_ms;
    }
    return spike;
}
```

Makefile

```
APP = my_model_curr_exp
```


Makefile

```
APP = my_model_curr_exp

# This is the folder where things will be built (this will be created)
BUILD_DIR = build/
```


Makefile

```
APP = my_model_curr_exp

# This is the folder where things will be built (this will be created)
BUILD_DIR = build/

# This is the list of objects that will make up your neural model, as well as
# the synaptic plasticity type (or lack of if using static_impl)
# TODO: Add any extra objects to be compiled
MODEL_OBJS = $(EXTRA_SRC_DIR)/neuron/models/neuron_model_my_model_curr_exp.o \
             $(SOURCE_DIRS)/neuron/plasticity/synapse_dynamics_static_impl.o
```

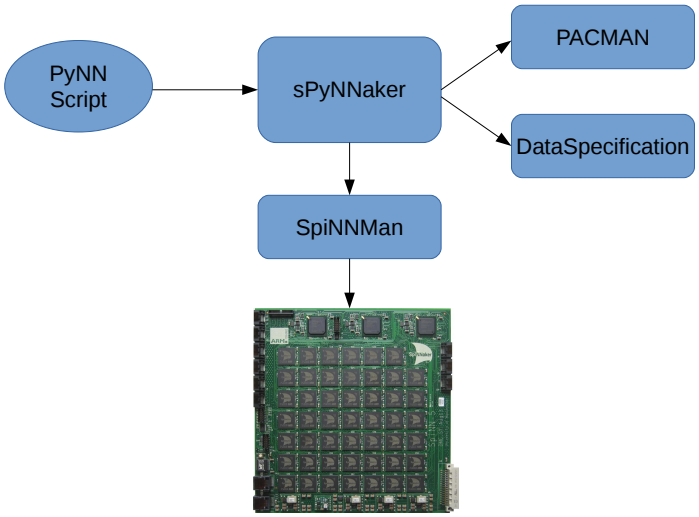

Makefile

```
APP = my_model_curr_exp

# This is the folder where things will be built (this will be created)
BUILD_DIR = build/

# This is the list of objects that will make up your neural model, as well as
# the synaptic plasticity type (or lack of if using static_impl)
# TODO: Add any extra objects to be compiled
MODEL_OBJS = $(EXTRA_SRC_DIR)/neuron/models/neuron_model_my_model_curr_exp.o \
             $(SOURCE_DIRS)/neuron/plasticity/synapse_dynamics_static_impl.o

# This is the header of the neuron model, containing the definintion of
# neuron_t
# TODO: Ensure this matches your neuron model header name
NEURON_MODEL_H = $(EXTRA_SRC_DIR)/neuron/models/neuron_model_my_model_curr_exp.h
```

Makefile

```
APP = my_model_curr_exp

# This is the folder where things will be built (this will be created)
BUILD_DIR = build/

# This is the list of objects that will make up your neural model, as well as
# the synaptic plasticity type (or lack of if using static_impl)
# TODO: Add any extra objects to be compiled
MODEL_OBJS = $(EXTRA_SRC_DIR)/neuron/models/neuron_model_my_model_curr_exp.o \
             $(SOURCE_DIRS)/neuron/plasticity/synapse_dynamics_static_impl.o

# This is the header of the neuron model, containing the definintion of
# neuron_t
# TODO: Ensure this matches your neuron model header name
NEURON_MODEL_H = $(EXTRA_SRC_DIR)/neuron/models/neuron_model_my_model_curr_exp.h

# This is the header containing the synapse shaping type
# (exponential in this case)
# TODO: Ensure that this is the desired shaping
SYNAPSE_TYPE_H = $(SOURCE_DIRS)/neuron/synapse_types/synapse_types_exponential_impl.h

include ../Makefile.common
```


Python Interface – Why?



Python Interface – Class Definition

```
from spynnaker.pyNN.models.abstract_models.abstract_population_vertex import \
    AbstractPopulationVertex


class IzhikevichCurrentExponentialPopulation(
        AbstractPopulationVertex
```


Python Interface – Class Definition

```
from spynnaker.pyNN.models.abstract_models.abstract_population_vertex import \
    AbstractPopulationVertex
from spynnaker.pyNN.models.abstract_models.abstract_model_components.\
    abstract_exp_population_vertex import AbstractExponentialPopulationVertex


class IzhikevichCurrentExponentialPopulation(
        AbstractPopulationVertex
        AbstractExponentialPopulationVertex):
```

Python Interface – max atoms

```
class IzhikevichCurrentExponentialPopulation(
        AbstractPopulationVertex
        AbstractExponentialPopulationVertex):

    _model_based_max_atoms_per_core = 255
```


Python Interface – initializer

```
class IzhikevichCurrentExponentialPopulation(
        AbstractPopulationVertex
        AbstractExponentialPopulationVertex):

    _model_based_max_atoms_per_core = 255

    def __init__(
            self, n_neurons, machine_time_step, timescale_factor,
            spikes_per_second, ring_buffer_sigma, constraints=None, label=None,
```


Python Interface – initializer

```
class IzhikevichCurrentExponentialPopulation(
        AbstractPopulationVertex
        AbstractExponentialPopulationVertex):

    _model_based_max_atoms_per_core = 255

    def __init__(
            self, n_neurons, machine_time_step, timescale_factor,
            spikes_per_second, ring_buffer_sigma, constraints=None, label=None,
            tau_syn_E=5.0, tau_syn_I=5.0,
```


Python Interface – initializer

```
class IzhikevichCurrentExponentialPopulation(
        AbstractPopulationVertex
        AbstractExponentialPopulationVertex):

    _model_based_max_atoms_per_core = 255

    def __init__(
            self, n_neurons, machine_time_step, timescale_factor,
            spikes_per_second, ring_buffer_sigma, constraints=None, label=None,
            tau_syn_E=5.0, tau_syn_I=5.0,
            a=0.02, b=0.2, c=-65.0, d=2.0, i_offset=0, v_init=-70.0, u_init=-14.0):
```

Python Interface – initializer

```
class IzhikevichCurrentExponentialPopulation(
        AbstractPopulationVertex
        AbstractExponentialPopulationVertex):

    _model_based_max_atoms_per_core = 255

    def __init__(
            self, n_neurons, machine_time_step, timescale_factor,
            spikes_per_second, ring_buffer_sigma, constraints=None, label=None,
            tau_syn_E=5.0, tau_syn_I=5.0,
            a=0.02, b=0.2, c=-65.0, d=2.0, i_offset=0, v_init=-70.0, u_init=-14.0):
        AbstractPopulationVertex.__init__(
            self, n_params=8,
```


Python Interface – initializer

```
class IzhikevichCurrentExponentialPopulation(
        AbstractPopulationVertex
        AbstractExponentialPopulationVertex):

    _model_based_max_atoms_per_core = 255

    def __init__(
            self, n_neurons, machine_time_step, timescale_factor,
            spikes_per_second, ring_buffer_sigma, constraints=None, label=None,
            tau_syn_E=5.0, tau_syn_I=5.0,
            a=0.02, b=0.2, c=-65.0, d=2.0, i_offset=0, v_init=-70.0, u_init=-14.0):
        AbstractPopulationVertex.__init__(
            self, n_params=8, n_global_params=1,
```


Python Interface – initializer

```
class IzhikevichCurrentExponentialPopulation(
        AbstractPopulationVertex
        AbstractExponentialPopulationVertex):

    _model_based_max_atoms_per_core = 255

    def __init__(
            self, n_neurons, machine_time_step, timescale_factor,
            spikes_per_second, ring_buffer_sigma, constraints=None, label=None,
            tau_syn_E=5.0, tau_syn_I=5.0,
            a=0.02, b=0.2, c=-65.0, d=2.0, i_offset=0, v_init=-70.0, u_init=-14.0):
        AbstractPopulationVertex.__init__(
            self, n_params=8, n_global_params=1, binary="IZK_curr_exp.aplx",
```


Python Interface – initializer

```
class IzhikevichCurrentExponentialPopulation(
        AbstractPopulationVertex
        AbstractExponentialPopulationVertex):

    _model_based_max_atoms_per_core = 255

    def __init__(
            self, n_neurons, machine_time_step, timescale_factor,
            spikes_per_second, ring_buffer_sigma, constraints=None, label=None,
            tau_syn_E=5.0, tau_syn_I=5.0,
            a=0.02, b=0.2, c=-65.0, d=2.0, i_offset=0, v_init=-70.0, u_init=-14.0):
        AbstractPopulationVertex.__init__(
            self, n_params=8, n_global_params=1, binary="IZK_curr_exp.aplx",
            max_atoms_per_core=(IzhikevichCurrentExponentialPopulation.
                                _model_based_max_atoms_per_core),
```

Python Interface – initializer

```
class IzhikevichCurrentExponentialPopulation(
        AbstractPopulationVertex
        AbstractExponentialPopulationVertex):

    _model_based_max_atoms_per_core = 255

    def __init__(
            self, n_neurons, machine_time_step, timescale_factor,
            spikes_per_second, ring_buffer_sigma, constraints=None, label=None,
            tau_syn_E=5.0, tau_syn_I=5.0,
            a=0.02, b=0.2, c=-65.0, d=2.0, i_offset=0, v_init=-70.0, u_init=-14.0):
        AbstractPopulationVertex.__init__(
            self, n_params=8, n_global_params=1, binary="IZK_curr_exp.aplx",
            max_atoms_per_core=(IzhikevichCurrentExponentialPopulation.
                                _model_based_max_atoms_per_core),
            n_neurons=n_neurons,
            machine_time_step=machine_time_step,
            timescale_factor=timescale_factor,
            spikes_per_second=spikes_per_second,
            ring_buffer_sigma=ring_buffer_sigma,
            label=label, constraints=constraints)
```


Python Interface – initializer

```
class IzhikevichCurrentExponentialPopulation(
        AbstractPopulationVertex
        AbstractExponentialPopulationVertex):

    _model_based_max_atoms_per_core = 255

    def __init__(
            self, n_neurons, machine_time_step, timescale_factor,
            spikes_per_second, ring_buffer_sigma, constraints=None, label=None,
            tau_syn_E=5.0, tau_syn_I=5.0,
            a=0.02, b=0.2, c=-65.0, d=2.0, i_offset=0, v_init=-70.0, u_init=-14.0):
        ...
        AbstractExponentialPopulationVertex.__init__(
            self, n_neurons=n_neurons, tau_syn_E=tau_syn_E,
            tau_syn_I=tau_syn_I, machine_time_step=machine_time_step)
```


Python Interface - Parameters

- Parameters can be:
  - Individual values
  - Array of values (one per neuron)
  - RandomDistribution
- Normalise Parameters
  - utility_calls.convert_param_to_numpy(param, n_neurons)

Python Interface – initializer

```
class IzhikevichCurrentExponentialPopulation(
        AbstractPopulationVertex
        AbstractExponentialPopulationVertex):

    _model_based_max_atoms_per_core = 255

    def __init__(
            self, n_neurons, machine_time_step, timescale_factor,
            spikes_per_second, ring_buffer_sigma, constraints=None, label=None,
            tau_syn_E=5.0, tau_syn_I=5.0,
            a=0.02, b=0.2, c=-65.0, d=2.0, i_offset=0, v_init=-70.0, u_init=-14.0):
        ...
        ...
        self._a = utility_calls.convert_param_to_numpy(a, n_neurons)
        self._b = utility_calls.convert_param_to_numpy(b, n_neurons)
        self._c = utility_calls.convert_param_to_numpy(c, n_neurons)
        self._d = utility_calls.convert_param_to_numpy(d, n_neurons)
        self._i_offset = utility_calls.convert_param_to_numpy(i_offset, n_neurons)
        self._v_init = utility_calls.convert_param_to_numpy(v_init, n_neurons)
        self._u_init = utility_calls.convert_param_to_numpy(u_init, n_neurons)
```

Python Interface – state initializers

```
class IzhikevichCurrentExponentialPopulation(
        AbstractPopulationVertex
        AbstractExponentialPopulationVertex):

    ...

    def initialize_v(self, v_init):
        self._v_init = utility_calls.convert_param_to_numpy(v_init, self.n_atoms)

    def initialize_u(self, u_init):
        self._u_init = utility_calls.convert_param_to_numpy(u_init, self.n_atoms)
```


Python Interface – properties

```
class IzhikevichCurrentExponentialPopulation(
        AbstractPopulationVertex
        AbstractExponentialPopulationVertex):

    ...

    @property
    def a(self):
        return self._a

    @a.setter
    def a(self, a):
        self._a = utility_calls.convert_param_to_numpy(a, self.n_atoms)

    @property
    def b(self):
        return self._b

    @b.setter
    def b(self, b):
        self._b = utility_calls.convert_param_to_numpy(b, self.n_atoms)

    ...
```


Python Interface – model name

```
class IzhikevichCurrentExponentialPopulation(
        AbstractPopulationVertex
        AbstractExponentialPopulationVertex):

    ...

    @property
    def model_name(self):
        return "IZK_curr_exp"
```


Python Interface – set max atoms

```
class IzhikevichCurrentExponentialPopulation(
        AbstractPopulationVertex
        AbstractExponentialPopulationVertex):

    ...

    @staticmethod
    def set_model_max_atoms_per_core(new_value):
        IzhikevichCurrentExponentialPopulation.\
            _model_based_max_atoms_per_core = new_value
```

Python Interface – cpu usage

```
class IzhikevichCurrentExponentialPopulation(
        AbstractPopulationVertex
        AbstractExponentialPopulationVertex):

    ...

    def get_cpu_usage_for_atoms(self, vertex_slice, graph):
        return 782 * ((vertex_slice.hi_atom - vertex_slice.lo_atom) + 1)
```


Python Interface – parameters

```
class IzhikevichCurrentExponentialPopulation(
    ...
    def get_parameters(self):
        return [
            # REAL a
            NeuronParameter(self._a, DataType.S1615),
            # REAL b
            NeuronParameter(self._b, DataType.S1615),
            # REAL c
            NeuronParameter(self._c, DataType.S1615),
            # REAL d
            NeuronParameter(self._d, DataType.S1615),
            # REAL v
            NeuronParameter(self._v_init, DataType.S1615),
            # REAL u
            NeuronParameter(self._u_init, DataType.S1615),
            # REAL I_offset
            NeuronParameter(self._i_offset, DataType.S1615),
            # REAL this_h
            NeuronParameter(self._machine_time_step / 1000.0, DataType.S1615)
        ]
```

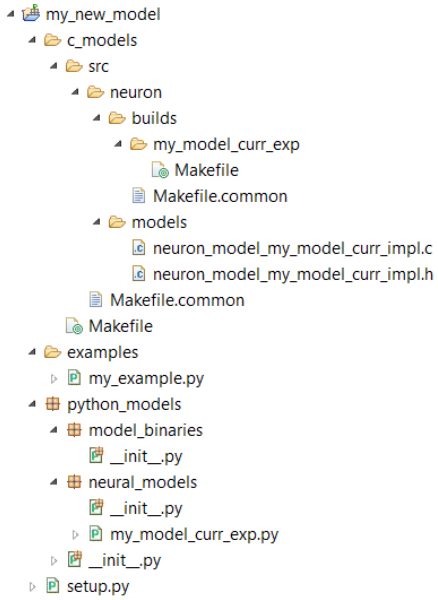

Python Interface – global params

```
class IzhikevichCurrentExponentialPopulation(
    ...
    def get_global_parameters(self):
        return [
            # REAL machine_timestep_ms
            NeuronParameter(self._machine_time_step / 1000.0, DataType.S1615)
        ]
```


Python Interface – abstract impl

```
class IzhikevichCurrentExponentialPopulation(
    ...
    def is_population_vertex(self):
        return True

    def is_exp_vertex(self):
        return True
```

New Model Template



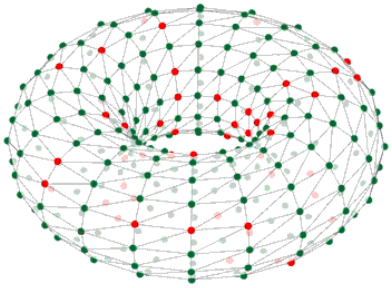
Using Your Model

```
import pyNN.spiNNaker as p
import python_models as new_models
```


Using Your Model

```
import pyNN.spiNNaker as p
import python_models as new_models

my_model_pop = p.Population(
    1, new_models.MyModelCurrExp,
    {"my_parameter": 2.0,
     "i_offset": i_offset},
    label="my_model_pop")
```

SpiNNaker System Software



Steve Temple
SpiNNaker Workshop – Manchester – Sep 2015

European Research Council
Established by the European Commission

Human Brain Project

EPSRC

SpiNNaker

Overview

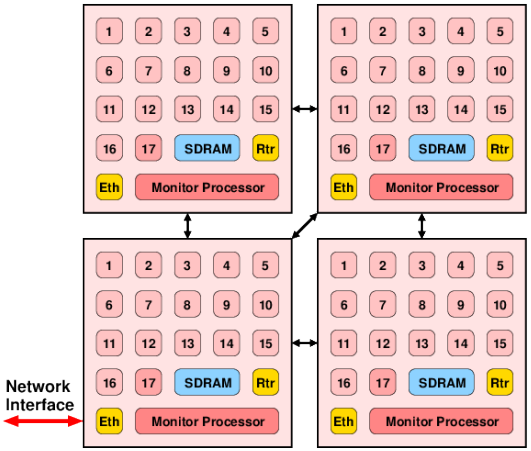
- SpiNNaker applications and their environment
- SC&MP, *ybug* and application loading
- SARK (SpiNNaker Application Runtime Kernel)
  - Application start-up
  - SARK function library
  - Examples
  - Documentation

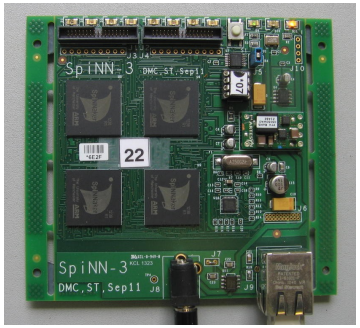
Please interrupt if you have a question!

Building Applications

- Languages – mostly C with bits of assembler
- Toolchain choice
  - ARM tools – RVDS 4 and DS-5 (free for academics)
  - GCC – Mentor Graphics Code Sourcery Lite (free)
- Library support
  - Toolchain libraries – C library functions, maths, etc
  - SARK – low-level SpiNNaker support library
  - Spin1 API – event-based application library
- Linking – support libs + application code
  - Creates application to be loaded
  - Application file format is APLX

Execution Environment (1)

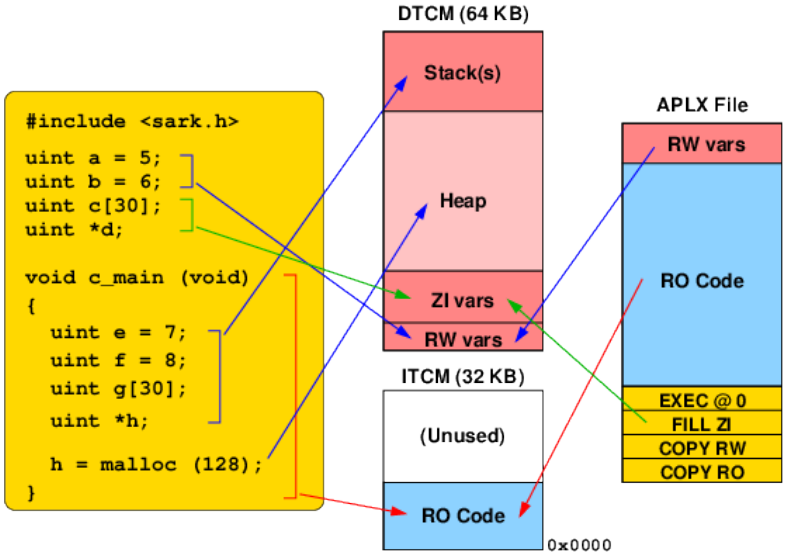




Execution Environment (2)

- One application per core
- Executable code (instructions) in ITCM (32 KB)
- Data (variables, stacks, heap) in DTCM (64 KB)
- Bulk and/or shared data in SDRAM (128 MB)
- Code/data access from ITCM/DTCM is fast (5 ns)
- Data access to SDRAM is slow (> 100 ns) and subject to contention
- DMA controller in each core can move bulk data between I/DTCM and SDRAM faster (~ 15 ns/word) without requiring CPU

Mapping Program to Memory



SC&MP

- "SpiNNaker Control & Monitor Program"
- Loaded onto all Monitor Processors during bootstrap
- Communicates with host computer using SCP (SpiNNaker Command Protocol) over SDP
- Supervises operation of a single chip
- Allows program loading to Application Cores
- Acts as router for SDP packets between any pair of cores or with external Internet endpoints
- Flashes the LED!

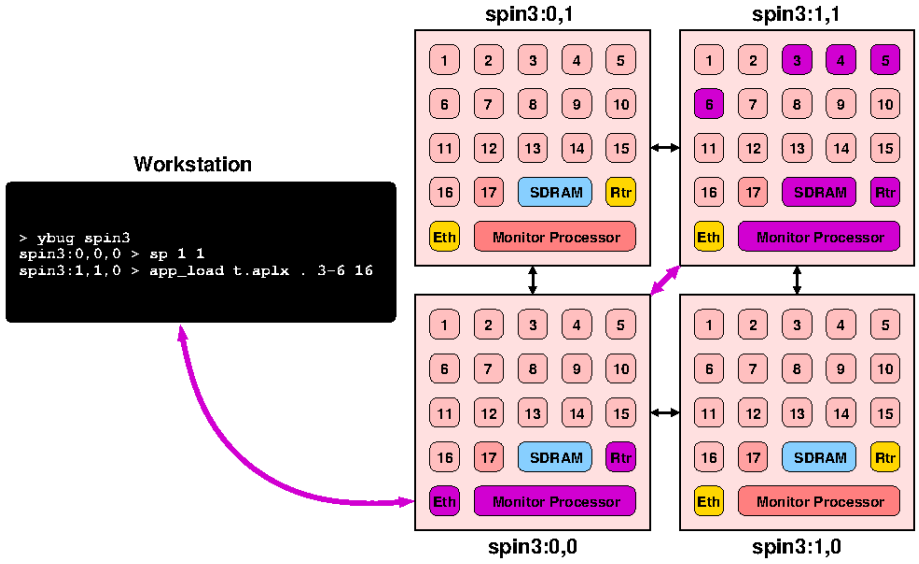
SC&MP, SCP and *ybug*

- SC&MP provides command interface via SCP
  - `Ver` – give S/W version, etc
  - `Read (addr, length)` – read SpiNNaker memory
  - `Write (addr, length, data)` – write SpiNNaker memory
  - `Reset (core_mask)` – reset Application Cores
- Host (workstation) embeds SCP/SDP in UDP/IP to talk to SpiNNaker Monitor Processor on the Root Chip
- *ybug* is a simple command-line tool which runs on a workstation and provides an interface to SC&MP for application loading and debug

Application Loading (1)

- *ybug* sends the application APLX to the relevant SpiNNaker chips.
- The APLX image is copied to a known place in shared memory
- *ybug* requests that the relevant Application cores are reset.
- The reset code is an *APLX loader* which loads the image according to instructions in the APLX header
- This usually results in the application being copied into ITCM and entered at address zero (the ARM reset vector)

Application Loading (2)



SARK

- SpiNNaker Application Runtime Kernel
- Three main functions
  1) Application start-up
  2) Library of useful functions
  3) Communication via SDP with Monitor Processor (and hence rest of system)
- SARK is automatically linked with applications when they are built
- Occupies around 2 KB in the image

Application Start-Up

- Start-up code at start of ITCM is SARK
  - Configures stacks for 4 ARM execution modes
  - Initialises Heap and SDP message buffers in DTCM
  - Initialises shared-memory data structure (VCPU)
  - Calls a function to do pre-application set-up
  - Calls the function **`c_main`**, the application entry point
  - Calls a function to do post-application clean-up
  - Goes to sleep!
- Some applications will never terminate
- SARK provides SDP communications with the application

SARK Library (1)

- CPU control
  - Interrupt disabling and enabling
  - Entering low power (sleep) mode
- Memory manipulation
  - Memory copy and fill (small footprint)
  - SDP message copying
- Pseudo-Random number generation (32-bit)
- SDP messaging
  - Message allocation in DTCM and shared memory
  - SDP message transmission

SARK Library (2)

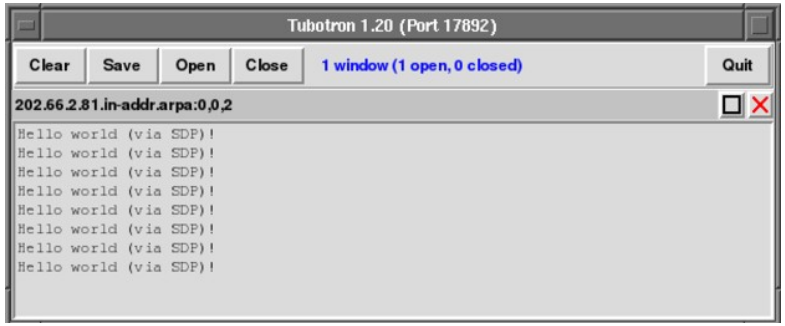
- Text output via “printf”
  - Text sent to a host system using SDP packets
  - Text buffered in SDRAM
- Hardware locks and semaphores
- Memory management
  - `malloc/free` for DTCM heap
  - `malloc/free` for shared memories (eg SDRAM) with locking
  - `malloc/free` for router MC routing table
- Environment queries
  - What is my core ID, chipID, etc

SARK Library (3)

- Hardware interfaces
  - LED control
  - Router control – setting MC and P2P table entries
  - VIC control – allocating interrupt handlers to specific hardware interrupts
- Timer management
  - Routines to schedule/cancel events at some time in the future
- Event management
  - Routines to associate events with interrupts
  - Management of priority event queues

SARK – Example 1

```
#include <sark.h>

void c_main (void)
{
  io_printf (IO_STD, "Hello world (via SDP)!\n");
  io_printf (IO_BUF, "Hello world (via SDRAM)!\n");
}
```



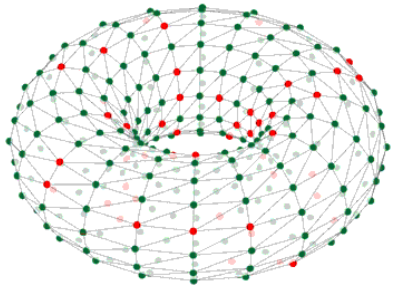
SARK - Example 2

```
#include <sark.h>

INT_HANDLER timer_int_han (void)
{
  tc[T1_INT_CLR] = (uint) tc;      // Clear interrupt in timer
  sark_led_set (LED_FLIP (1));     // Flip a LED
  vic[VIC_VADDR] = (uint) vic;     // Tell VIC we're done
}

void timer_setup (uint period)
{
  tc[T1_CONTROL] = 0xe2;               // Set up count-down mode
  tc[T1_LOAD] = sark.cpu_clk * period; // Load time (us)
  sark_vic_set (SLOT_0, TIMER1_INT, 1, timer_int_han);
}

void c_main ()
{
  io_printf (IO_STD, "Timer interrupt example\n");
  timer_setup (500000);       // (0.5 secs)
  cpu_sleep ();               // Send core to sleep
}
```

Documentation & Help

- SARK – notes in 1.30 Software release - `sark.pdf`
- *ybug* – user guide in 1.30 Software release - `ybug.pdf`
- `"spinnaker.h"` - describes the SpiNNaker hardware – memory maps, peripheral registers...
- `"sark.h"` describes all SARK data structures and functions. Commented in Doxygen style.
- All source code is provided...
- If desperate, talk to us!

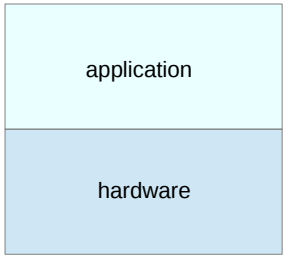
SpiNNaker API

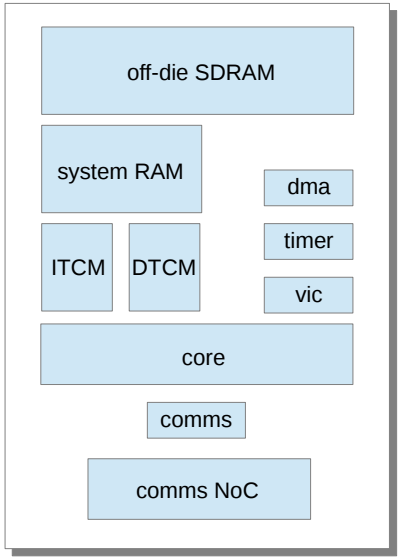


Luis Plana

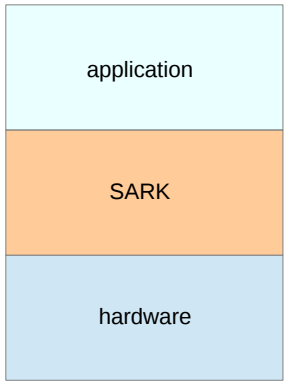
SpiNNaker Workshop, September 2015

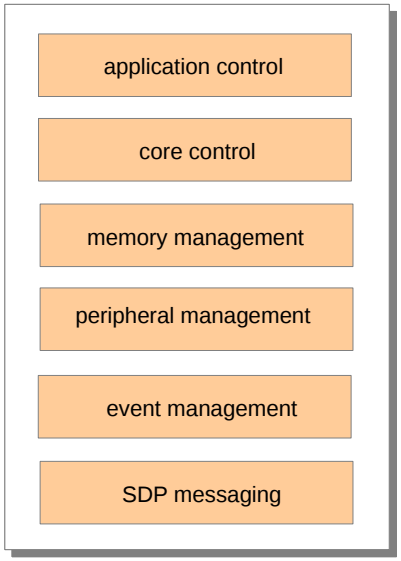
hardware resources



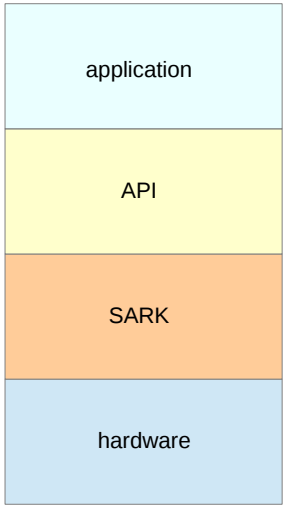


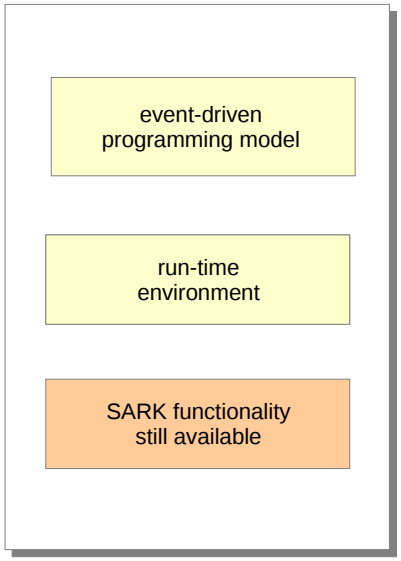
SARK: low-level software





API: run-time environment





event-driven model

applications do not control execution flow

applications indicate functions to be executed when events of interest occur

API controls execution and schedules application functions when appropriate

application functions are known as callbacks

events and callbacks

event	trigger
timer tick	periodic event has occurred
multicast packet received	multicast packet has arrived
DMA transfer done	scheduled DMA transfer completed successfully
SDP packet received	SDP packet has arrived
user event	application-triggered event has occurred

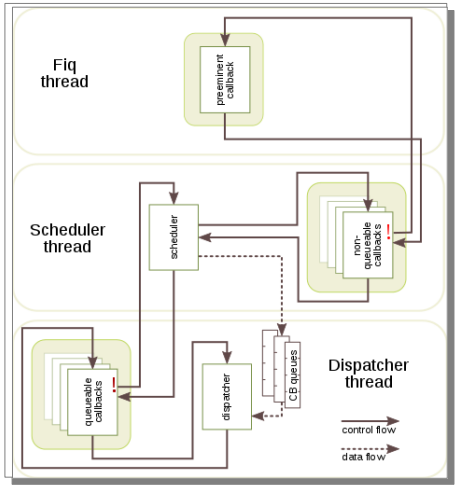
event	first argument	second argument
timer tick	simulation time (ticks)	null
MCP w/o payload received	key	0
MCP with payload received	key	payload
DMA transfer done	transfer ID	tag
SDP packet received	*mailbox	destination port
user event	arg0	arg1

priorities

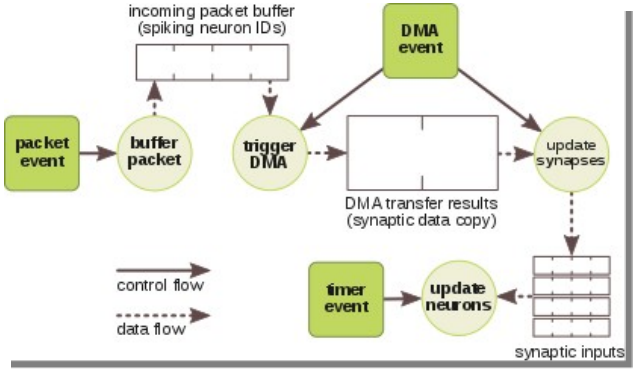
priority level = -1
only one callback
cannot be pre-empted

priority level = 0
can only be pre-empted
by priority -1 callback

priority level > 0
can be pre-empted
by priority <= 0 callbacks
scheduled in priority order



example: spiking neural network



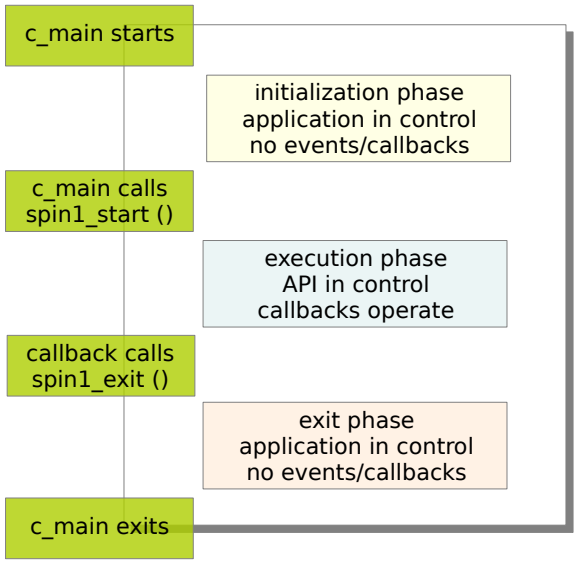
what is a sensible choice of priorities?

additional support

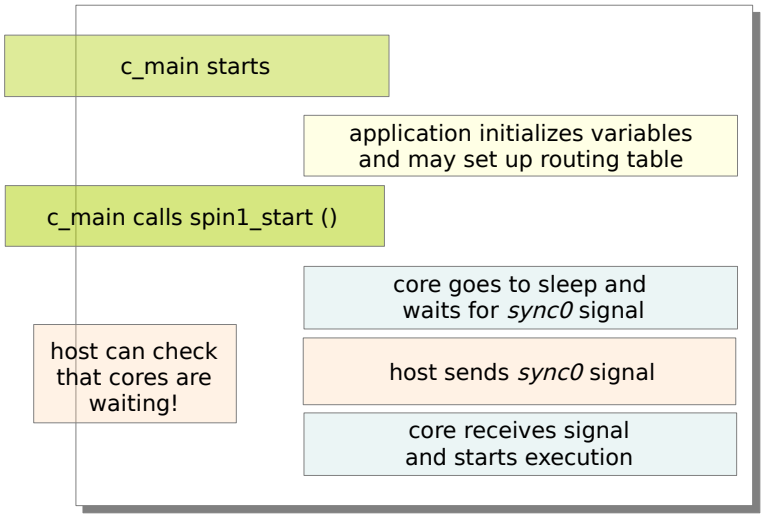
function	use
start/stop execution	start and stop simulation
set timer period	real-time or periodic callback
send multicast packet	inter-core communications
send SDP packet	host or I/O peripheral communications
start DMA transfer	software-managed cache
trigger user event	start a callback with priority <= 0
schedule callback	start a callback with priority > 0
enable/disable interrupts	critical section access (inter-thread control)
provide chip address and core ID	find out who you are
configure multicast routing table	setup routing entries

see API documentation for complete list

program structure



synchronization barrier



example: first.c

c_main

```
void c_main()
{
  // initialize variables and state
  // -------------------------------------------
  my_core = spin1_get_core_id();    // this core's id
  my_key  = ROUTING_KEY (my_core);  // this core's multicast routing key

  // initialize state in tubogrid
  // -------------------------------------------
  spin1_delay_us (100 * my_core);   // skew accesses to tubogrid!
  io_printf (IO_STD, state_colour[my_state]);

  // initialise routing tables: only one core needs to do it!
  // -------------------------------------------
  if (leadAp)
  {
    routing_table_init ();
  }

  // prepare for execution
  // -------------------------------------------
  // set timer tick value
  spin1_set_timer_tick (TIMER_TICK_PERIOD);

  // register callbacks
  spin1_callback_on (TIMER_TICK, update, 0);
  spin1_callback_on (MC_PACKET_RECEIVED, receive_packet, -1);

  // go
  // -------------------------------------------
  spin1_start(SYNC_WAIT);
}
```

example: first.c

definitions

```
// -------------------------------------------------
// simulation parameters and constants
// -------------------------------------------------
// set the execution speed and length
#define TIMER_TICK_PERIOD  125000   // 0.125s tick period (in microseconds)
#define TIMEOUT            128      // run for this many ticks

// indicate which core follows each other in the chain
uint next_core[] =
{
  2, 3, 4, 8, 6, 7, 11, 12, 5, 1, 10, 16, 9, 13, 14, 15
};

// enumerate possible core states
enum state_e {white = 0, red, green, blue};
typedef enum state_e state;

// an easy way to update to the next state
state next_state[] = {red, green, blue, white};

// prepare the strings to represent each state in tubogrid
char* state_colour[] =
{
  "#white;#fill;\n",
  "#red;#circle;\n",
  "#green;#circle;\n",
  "#blue;#circle;\n"
};
```

example: first.c

initialization

```
void routing_table_init ()
{
  // request table entries
  uint e = rtr_alloc (16);

  // write routing table entries to the router
  if (e != 0)
  {
    for (uint i = 0; i < 16; i++)
    {
      rtr_mc_set (e + i,                          // entry
                  ROUTING_KEY (i + 1),            // key
                  ROUTE_MASK,                     // mask
                  ROUTE_TO_CORE (next_core[i])    // route
                 );
    }
  }
}
```

example: first.c

packet callback

```
void receive_packet (uint pkt_key, uint pkt_payload)
{
  // update my state
    my_state = next_state[my_state];
}
```

example: first.c

timer callback

```
void update (uint ticks, uint b)
{
  // somebody has to get the changes started
  if ((ticks == 1) && (my_core == 1))
  {
    my_state = next_state[my_state];
  }

  // update if not finishing
  if (ticks <= TIMEOUT)
  {
    // check if state changed
    if (my_state != old_state)
    {
      // update tubogrid,
      io_printf (IO_STD, state_colour[my_state]);

      // send a packet to next core in the chain,
      spin1_send_mc_packet(my_key, 0, NO_PAYLOAD);

      // and remember state
      old_state = my_state;
    }
  }
  else
  {
    // finish
    spin1_exit (0);
  }
}
```

MANCHESTER 1824

to think about: pitfalls

asynchronous operation and communications

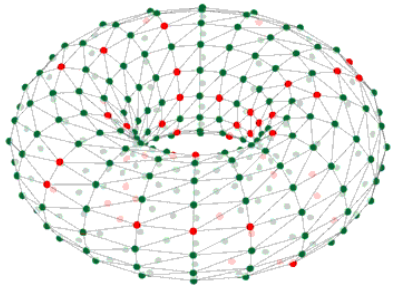
multicast packets can be dropped due to congestion

UDP-based I/O *not guaranteed!*

no floating-point support use fixed-point arithmetic

no globally-shared resources use message passing

Event-Driven Neural Simulation



Alex Rast

SpiNNaker Workshop, September 2015

European Research Council

Human Brain Project

EPSRC

SpiNNaker

Session Outline

1. What happens When a Simulation Runs?
- a. Design Assumptions
- b. Tool Chain Instantiation
- c. On-System Startup
- d. Real-Time Execution

2. Responding to Events
- a. Packet Received
- b. User Event
- c. DMA Done
- d. Timer Tick

3. Time and Events
- a. Event Priorities
- b. What is "Real-Time"?
- c. Adjusting Time Resolution
- d. Using Retarded Time

4. So how do You Make a Working Model?
- a. Handling Events
- b. Memory Utilisation
- c. Debugging and (Lack of) Visibility
- d. What SpiNNaker Can Do

What Happens When A Simulation Runs?

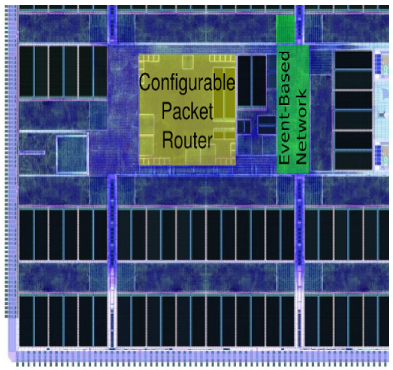


Fig1: SpiNNaker Chip

Design Assumptions

1. Memory: Cores only need access their own *limited* local memory. No global memory.

2. Communications: Via AER spikes only, multicast source-routed.

3. Event Rates: Real-time at "biologically meaningful" resolution. Hardware is much faster than any simulation time scale.

4. Model Dynamic Complexity: Very simple models are good enough. Most biological minutiæ don't matter.

5. Time Model: Execution is event-driven; time "models itself" (is implicit).

What Happens When A Simulation Runs?

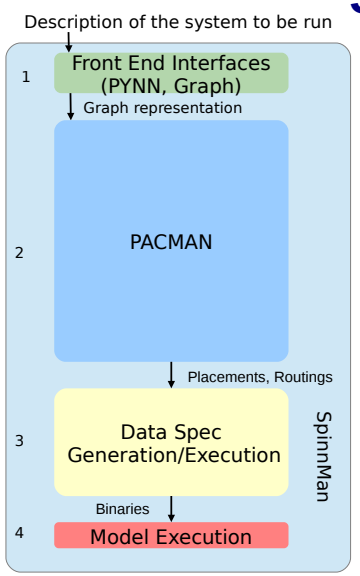


Fig2: The Tool Chain stack

Stage 1: Instantiation Through Tool Chain

1. Script binds to SpiNNaker front-end

2. Front-end converts the script into a set of nodes and edges

3. PACMAN partitions the nodes to cores and edges to routing table info and synaptic matrices in SDRAM

4. DSG transforms the partitioned graph into a set of instantiation specifications

5. DSE unpacks the specification into an on-chip executable

What Happens When A Simulation Runs?

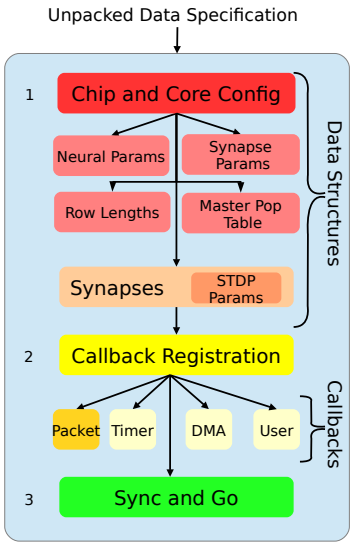


Fig3: Application Startup

Stage 2: On-System Startup

1. Generation of Core Data Structures:

- Neural Parameter Structures
- Synaptic Parameter Structures
- Synaptic Row Lengths
- Master Population Table
- Synaptic Connection Matrix
- STDP Parameters (if any)

2. Registering Callbacks

- Spike Received (buffer and ask for a DMA read)
- Timer Interrupt (start processing the next state update)
- DMA Complete (dump inputs into ring buffers and update SDTP)
- User Event (retrieve synapses from SDRAM)

3. Wait for Synchronisation, then Go!

What Happens When A Simulation Runs?

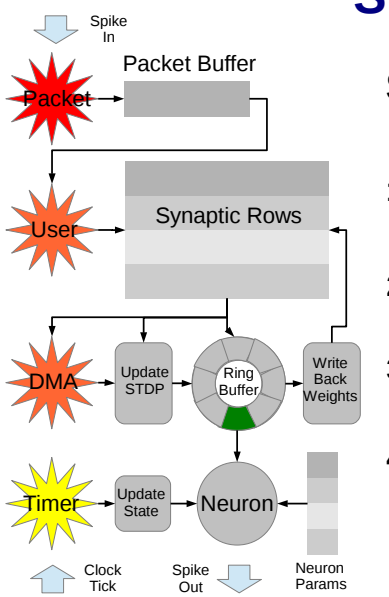


Fig4: Execution Model

Stage 3: Real-Time Execution

1. Packet Received [High Priority]:

2. User Event [Normal] (Request DMA)

3. DMA Completed [Normal]

4. Timer Tick [Low Priority]

Responding to Events

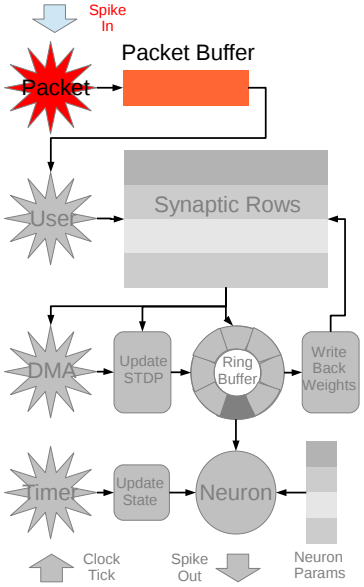


Fig4a: Execution Model

Packet Received

a) Dump packet into a buffer

b) Ask for a User event, if necessary

Responding to Events

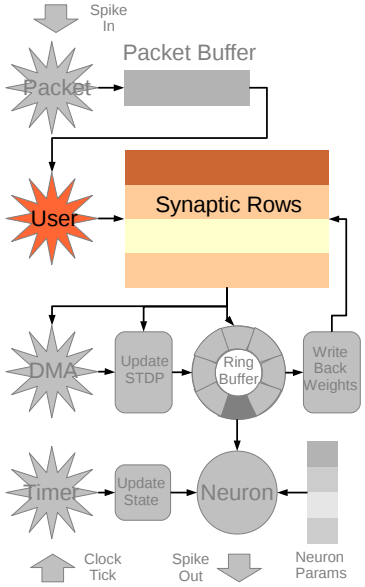


Fig4b: Execution Model

User Event

a) Retrieve next packet from buffer

b) Look up row address (in Master Population Table)

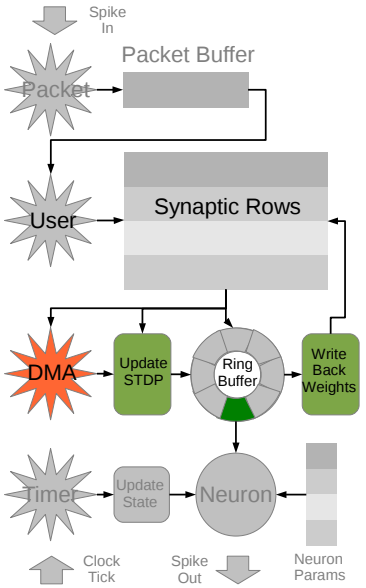
c) Set up the DMA in the controller

d) Start the next DMA transfer

e) Swap DMA buffers (for next transfer)

Responding to Events

DMA Completed



For each target in the row:

a) Start next DMA (everything under User Event)

b) Inject the current weight into the ring buffer at the delay indicated by the row's delay field

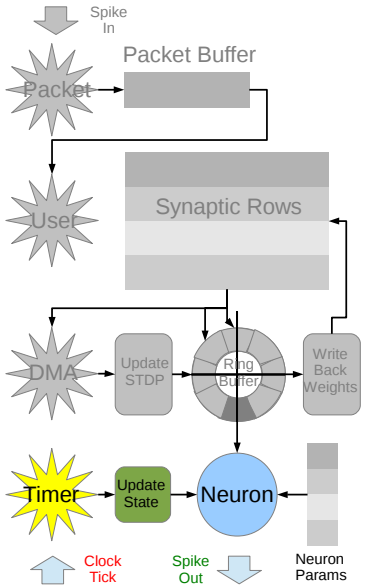
c) Update STDP, if enabled

d) Write back weight values to SDRAM via DMA

Fig4c: Execution Model

Responding to Events

Timer Tick



For each neuron on the core:

a) Decay the ring-buffer entries

b) Inject the current ring-buffer entry onto the neuron

c) Perform the neural state update

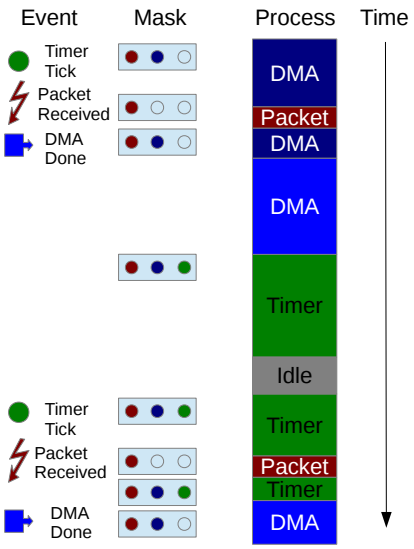
d) If neuron has reached threshold, spike.

e) If STDP is enabled, update for any post-synaptic spikes.

Fig4d: Execution Model

Time And Events

Event Priorities



Why? Events can overlap. Some events (packet received!) are critical. Priorities manage which events are serviced when.

Priority -1: Override priority. Can only assign to one event. MUST be serviced immediately. Assigned to packet received.

Priority ≥ 0: "Normal" priority. Maskable events with various priority levels. Assigned to all other events: DMA done (0), User (0), Timer (2)

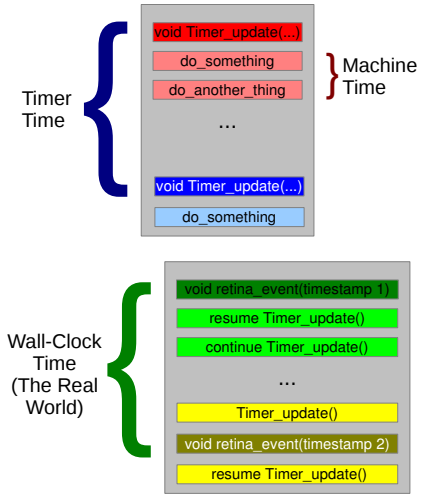
How? Set up in API when callback registered for event using

```
spin1_callback_on(event, callback, priority)
```

Fig5: Event Priority and Interrupt Servicing

Time And Events

What is "Real-Time"?



Machine Time: Core clock time. Unique to each core; NOT system-global. Intervals much smaller than Timer or "real-time".

Timer Time: Time between Timer ticks. Typically 1 ms; can be changed with an API call. Speedable-up or slowable relative to "real time".

Wall-Clock Time: External reference time. May be used by external devices (e.g. robots). Real-world "real-time".

Fig6: Different Time Domains

Time And Events

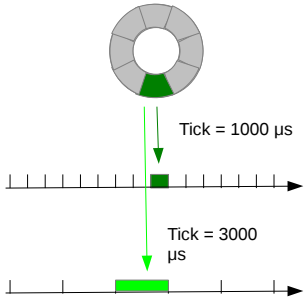


Fig7: Different Time Resolutions

Adjusting Time Resolution

Units: Timer resolution is currently in microsecond units

API: spin1_set_timer_tick() sets the time resolution (in µs).

Tool Chain: machineTimeStep in [Machine] section of spynnaker.cfg sets the time resolution (in µs).

Time And Events

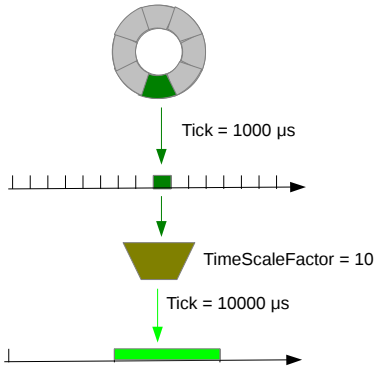


Fig8: Retarded Time

Using Retarded Time

Timescale Factor (F): Scales down Timer time so that *n* timer ticks with machine time step *m* µs corresponds to *n*m**F real-world µs

Toolchain Only: Not reflected in running code on the machine; this applies the time scaling through the toolchain itself.

In spynnaker.cfg: set with timeScaleFactor in the [Machine] section.

So How do You Make a Working Model?

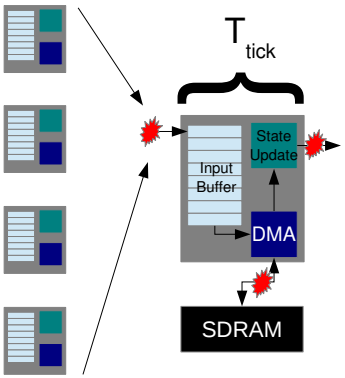


Fig9: Sources of Congestion

Handling Events

Unload the Fabric ASAP: Make the FIQ for packet-received efficient to prevent packet drops.

Issue Output Events As They Happen: Buffering output spikes only leads to bursty traffic and congestion.

Keep Time Resolution Coarse: Smaller timesteps drastically narrow the event receipt window.

DMA in Large Blocks: The controller is optimised for ~2k block size; small blocks may require a greater number of more inefficient transfers (DMA interrupts).

Slow Time, If Necessary: Slowing the time from real-time gives more slack for events to complete.

So How do You Make a Working Model?

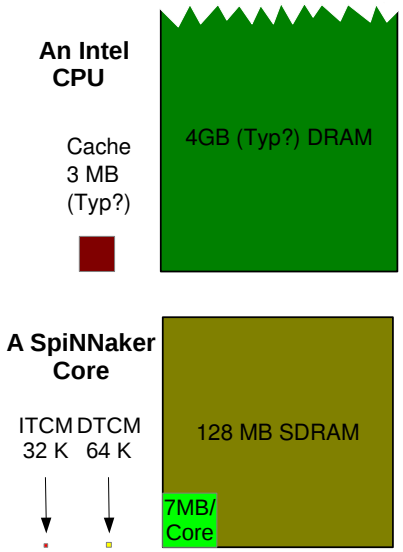


Fig10: What Cores See

Memory Limitations

Per Core:

64KB DTCM: All the neural parameters plus synaptic ring buffers PLUS temporary variables, lookup tables, etc. must fit in this space.

32KB ITCM: All the running code for a given model must fit in this space. This includes the SARK RTOS and the SpiNNaker API.

Per Chip:

128MB SDRAM: Partitioned amongst working cores. Synaptic weights, delays, and timestamps plus some system variables must fit here.

So How do You Make a Working Model?

Debugging and (Lack of) Visibility

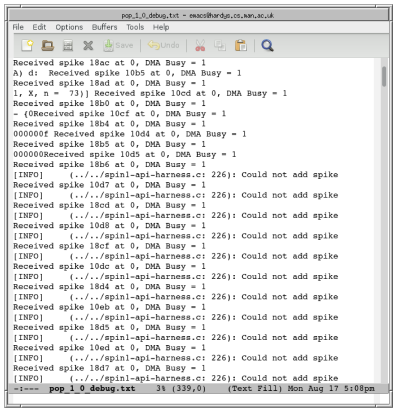


Fig11: Debug Output

There is No Global State: Cores run in different time domains. It is not only infeasible to set a "global breakpoint", it is meaningless. *Inspect* one core at a time.

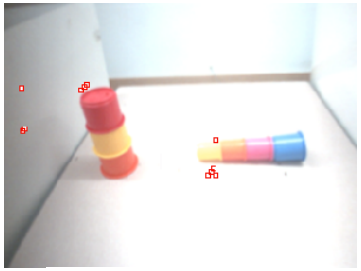
Interchip Timing Matters: Single-chip or -core results may not reveal everything. *Test* across multichip simulations to expose asynchronous bugs.

Debug Statements Alter Timing: Debug statements change the time events arrive and can cause breaking code to run. Use them, but be aware of the risks.

Events Interact: Because of different event priorities, some events may interrupt others in progress.

So How do You Make a Working Model?

What SpiNNaker Can Do



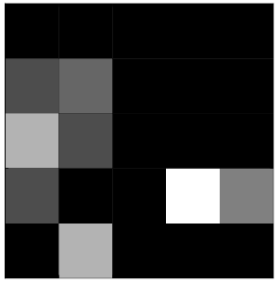


Fig12: A Model With Real-World Input

Experimenting With Network Parameters: Small simulations run fast enough that several runs may not just tune parameters but reveal patterns that show what parameters mean.

Scaling Networks: Large-scale networks can be run that simply would take too long to simulate even on a substantial cluster.

Exploring Network Design: Beyond parameter scaling, it is possible to alter network structure radically or even underlying neural models and still run in real-time.

Real-World Networks: By integrating with external hardware or robots, it is possible to explore real-world behaviour of large-scale networks in live situations.

5th SpiNNaker Workshop

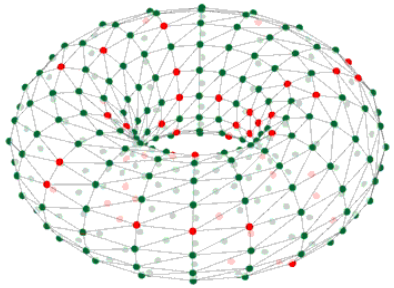
Day 4

September 10th 2015

Time	Session	Owner
09:00	Maths & fixed point libraries	Michael
09:45	Adding new models of synaptic plasticity	Jaime
10:30	Lab time (with coffee from 10:30)	
12:00	Lunch	
13:00	Connecting to external devices	Alex/Sergio
13:45	Lab time (with coffee at 15:00)	
16:00	Debugging using YBUG & GDB	Steve
17:00	Close	

Manchester, UK

Fixed-Point Maths and Libraries



Michael Hopkins

SpiNNaker Workshop, September 2015

Overview

1. Numerical calculation on SpiNNaker
2. ISO/IEC 18037 types and operations
3. A simple example
4. Some practical considerations
5. Libraries currently available
6. An example using the libraries
7. Using fixed-point to solve ODEs
8. Future directions

Numerical calculation on SpiNNaker

- No floating point hardware on SpiNNaker
- Software floating point available but too slow for most use cases (and larger binaries)
- Until recently, has needed hand-coded fixed point types and manipulations
- This approach not transparent so can be prone to maintenance issues & mysterious bugs
- More difficult than necessary for developers to translate algorithms into source code
- ISO draft 18037 for fixed point types and operations seen as a good solution

ISO 18037 types and operations

- Draft standard for native fixed point types & operations used like integer or floating point
- Currently only available on GNU toolchain >= 4.7 and ARM target architecture
- 8-, 16-, 32 and 64-bit precisions all available in (un-)saturated and (un-)signed versions
- *accum* type is 32-bit 'general purpose real'; we support io_printf() with s16.15 & u16.16
- *fract* type is 16-bit in [0,1]; we support io_printf() with s0.15 & u0.16

Operations supported are:

- prefix and postfix increment and decrement operators (++, --)
- unary arithmetic operators (+, -, !)
- binary arithmetic operators (+, -, *, /)
- binary shift operators (<<, >>)
- relational operators (<, <=, >=, >)
- equality operators (==, !=)
- assignment operators (+=, -=, *=, /=, <<=, >>=)
- conversions to and from integer, floating-point, or fixed-point types

A simple example

```
#include <stdfix.h>

#define REAL accum
#define REAL_CONST( x ) x##k

REAL  a, b, c = REAL_CONST( 100.001 );
accum d = REAL_CONST( 85.08765 );

int c_main( void )
{
   for( unsigned int i = 0; i < 50; i++ ) {

      a = i * REAL_CONST( 5.7 );

      b = a – i;

      if( a > d ) c  = a + b;
      else        c -= b;

      io_printf( IO_STD,
                 "\n i %u  a = %9.3k  b = %9.3k  c = %9.3k", i, a, b, c );
      }

   return 0;
}
```

Some practical considerations

- Range & precision e.g. for *accum* (s16.15) must have 0.000031 <= | x | <= 65536
- Still need to avoid divides in loops as these are slow on ARM architecture
- *saturated* types safe from overflow but significantly slower
- Need to remember that numerical precision is absolute rather than relative
- Literal constants require type suffix – simplest way is via macro REAL_CONST()
- Don't forget to `#include <stdfix.h>`
- Disciplined use of REAL and REAL_CONST() macros can parameterise entire code base
- Be careful to use the correct type suffix otherwise floating-point will be assumed

Libraries currently available - 1

1) *random.h* – suite of pseudo random number generators by MWH

Provides three high quality uniform generators of *uint32_t* values; Marsaglia's KISS 32 and KISS 64 and L'Ecuyer's WELL1024a.

- All three 'pass' the very stringent DIEHARD, dieharder and TestU01 test suites
- Trade-offs between speed, cycle length and equi-distributional properties
- Available in both simple-to-use form and with full user control over seeds

Have used these Uniform PRNGs as the basis for a set of Non-Uniform PRNGs including currently the following distributions:

- Gaussian
- Poisson (optimised for small rates at the moment)
- Exponential

...with more on the way. Let us know your requirements and we will try to help.

Libraries currently available - 2

2) *stdfix-full-iso.h* & *stdfix-math.h* – ISO & transcendental functions by DRL

Fill in the gaps in the GCC implementation of the ISO draft fixed point maths standard and some extensions:

- Standardised type conversions between fixed point representations
- Utility functions for all types i.e. abs(x), min(x), max(x), round(x), countls(x)
- Mechanism for automatically inferring the right argument type (uses GNU extension)

Fixed point replacements for essential floating point *libm* functions i.e. expk(x), sqrtk(x), logk(x), sink(x), cosk(x) and others such as atank(x), powk(x,y), 1/x on the way

- Hand-optimised for speed and accuracy on ARM architecture
- 10-30x faster than *libm* calls, hence feasible for use inside loops if necessary

An example using the libraries

```
accum           a, b, c, d;
uint32_t        r1;
unsigned fract  uf1;

init_WELL1024a_simp();  // need to initialise WELL1024a RNG before use

for( unsigned int i = 0; i < 22; i++ ) {

    r1 = WELL1024a_simp();                    // draw from Uniform RNG

    uf1 = (unsigned fract) ulrbits( r1 );     // convert to unsigned fract

// draw from Std Gaussian distribution using MARS64
    a = gaussian_dist_variate( mars_kiss64_simp, NULL );

// do some calculations on a and then log()
    b = logk( absk( a * REAL_CONST( 100.0 ) ) );

// sqrt() of value drawn from Exponential distribution using WELL1024a
    c = sqrtk( exponential_dist_variate( WELL1024a_simp, NULL ) );

    d = expk( (accum) ( i - 10 ) );       // exp() from -10 to 11

    io_printf( IO_STD, "\n i %4u
      uf1=[Uniform{*}]= %8.6R  a=[Gauss{*}]= %7.3k b=[ln(abs(100 a))]= %7.3k
      c=[sqrt(Exponential{*})]= %7.3k  d=[exp(i-10)]= %10.3k ", i, uf1, a, b, c, d );

    }
```

Using fixed-point to solve ODEs - 1

- Simulating neuron models usually means solving Ordinary Differential Equations (ODEs)
- This ranges from very easy (current input LIF has simple closed-form) solution to very challenging i.e. Hodgkin-Huxley with 4 state variables, nonlinear and very 'stiff' ODE
- Numerical calculations are required with a balance between accuracy & efficiency
- With care and attention to detail, fixed-point can be used to get very close to floating-point results. However, models with more complex behaviour are a significant challenge
- A new approach called *Explicit Solver Reduction* (ESR) makes this easier in many cases and is described in an upcoming paper: Hopkins & Furber (2015), “Accuracy and Efficiency in Fixed-Point Neural ODE Solvers” , *Neural Computation* **27**, 1–35
- Good results found for Izhikevich neuron at real-time simulation speed & 1 ms time step

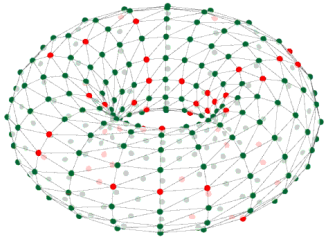
Using fixed-point to solve ODEs - 2

```
/*
   ESR algebraic reduction of the combination of Izhikevich neuron model and
   Runge-Kutta 2nd order midpoint method.  Hand-optimised interim variables and
   arithmetic ordering for balance between speed and accuracy.  See Neural Computation
   paper for more details.
*/
static inline void _rk2_kernel_midpoint( REAL h, neuron_pointer_t neuron,
                                         REAL input_this_timestep ) {
// to match Mathematica names
    REAL lastV1 = neuron->V;
    REAL lastU1 = neuron->U;
    REAL a = neuron->A;
    REAL b = neuron->B;

// generate common interim variables
    REAL pre_alph = REAL_CONST(140.0) + input_this_timestep - lastU1;
    REAL alpha = pre_alph
                 + ( REAL_CONST(5.0) + REAL_CONST(0.0400) * lastV1 ) * lastV1;
    REAL eta = lastV1 + REAL_HALF( h * alpha );

// could be represented as a long fract but need efficient mixed-arithmetic functions
    REAL beta = REAL_HALF( h * ( b * lastV1 - lastU1 ) * a );

// update neuron state
    neuron->V += h * ( pre_alph - beta
                       + ( REAL_CONST(5.0) + REAL_CONST(0.0400) * eta ) * eta );

    neuron->U += a * h * ( -lastU1 - beta + b * eta );
}
```

Future directions

- Optimise operations on differing fixed point types i.e. *accum * long fract*
- Add to *stdfix-math* (e.g. new argument types and special functions)
- Add to *random* (e.g. longer cycle uniform PRNG and more non-uniform distributions)
- New libraries such as probability distributions to allow Bayesian inference tools
- io_printf() to be extended to more types such as *long fract*, *unsigned long fract*
- Linear Algebra operations such as matrix multiply, SVD and other decompositions
- SpiNNaker architecture potentially good choice for massively parallel algorithms e.g. MCMC

Adding new models of synaptic plasticity



Jamie Knight

SpiNNaker Workshop
September 2015

European Research Council
Established by the European Commission

Human Brain Project

EPSRC

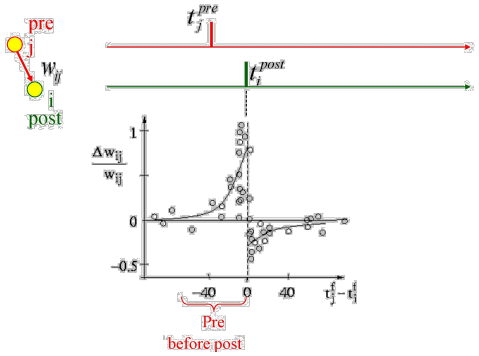
SpiNNaker

Outline

- Introduction to spike-timing dependent plasticity
- Simulating STDP
- Limitations of pair-based STDP
- Triplet STDP
- SpiNNaker implementation

Introduction to spike-timing dependent plasticity

"Cells that fire together, wire together"



http://www.scholarpedia.org/article/Spike-timing_dependent_plasticity

Simulating STDP - Traces

Pre-synaptic trace

$$\frac{dx_j}{dt} = -\frac{x_j}{\tau_x} + \sum_{t_j^f} \delta(t - t_j^f)$$

Post-synaptic trace

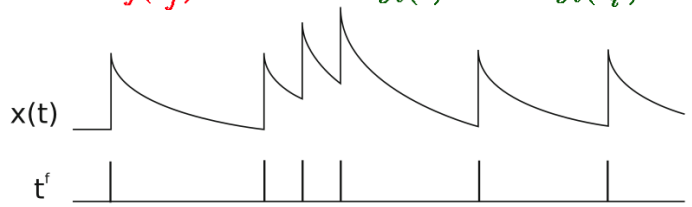
$$\frac{dy_i}{dt} = -\frac{y_i}{\tau_y} + \sum_{t_i^f} \delta(t - t_i^f)$$

At pre-synaptic spike time

$$x_j(t) = 1 + x_j(t_j^f) e^{-\frac{t - t_j^f}{\tau_x}}$$

At post-synaptic spike time

$$y_i(t) = 1 + y_i(t_i^f) e^{-\frac{t - t_i^f}{\tau_y}}$$



Morrison, A., Diesmann, M., & Gerstner, W. (2008). Phenomenological models of synaptic plasticity based on spike timing. Biological Cybernetics, 98, 459–478.

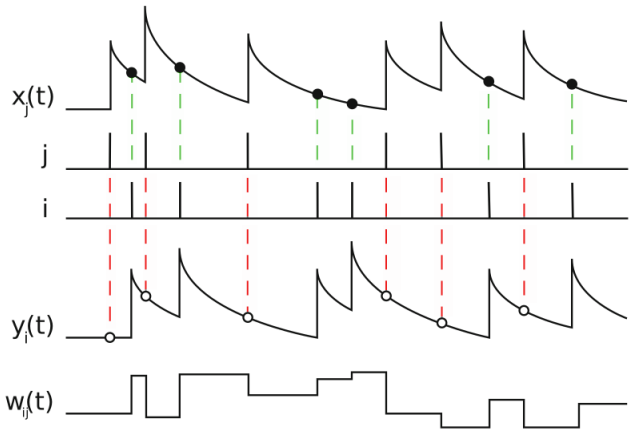
Simulating STDP - Weight update

Pre-synaptic weight update

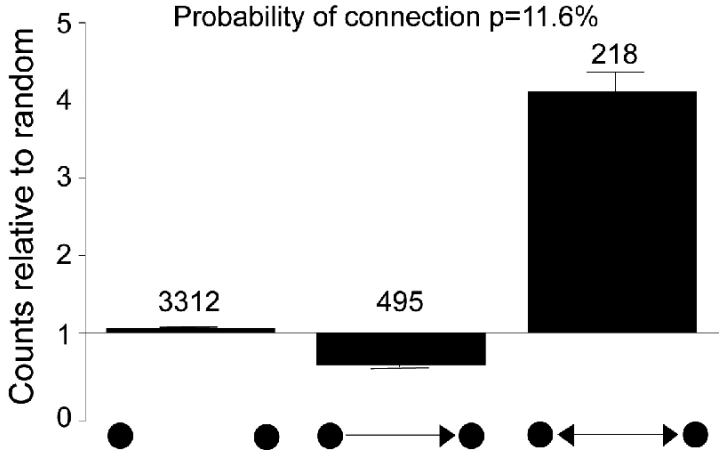
$$\Delta w_{ij}^{-} = F_{-}(w_{ij}) y_i(t_j^f)$$

Post-synaptic weight update

$$\Delta w_{ij}^{+} = F_{+}(w_{ij}) x_j(t_i^f)$$

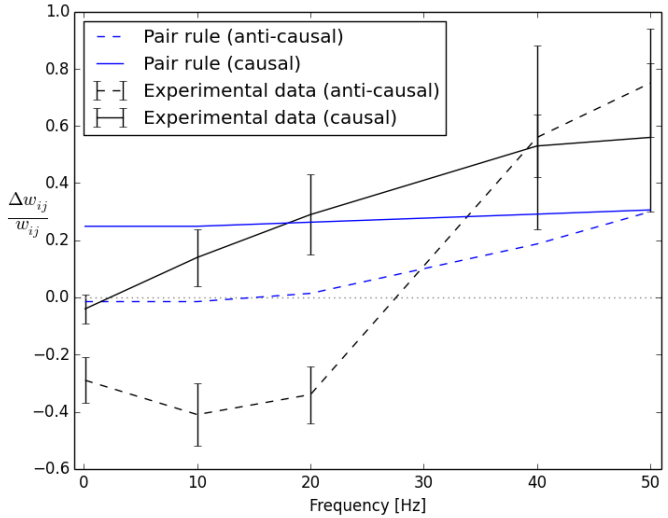


Limitations of pair-based STDP



Song, S., Sjöström, P. J., Reigl, M., Nelson, S., & Chklovskii, D. B. (2005). Highly nonrandom features of synaptic connectivity in local cortical circuits. PLoS Biology, 3(3), 0507–0519.

Limitations of pair-based STDP



Sjöström, P. J., Turrigiano, G. G., & Nelson, S. B. (2001). Rate, timing, and cooperativity jointly determine cortical synaptic plasticity. *Neuron*, *32*(6), 1149–64.

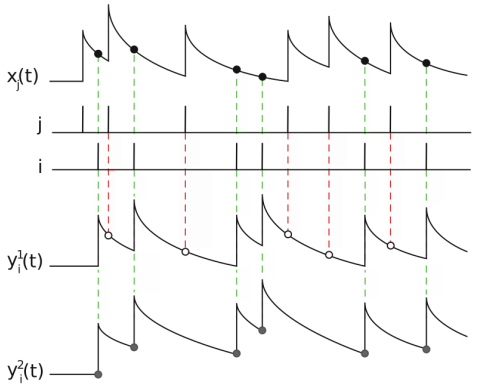
Triplet STDP

Slow post-synaptic trace

$$y_i^2(t) = \left(1 + y_i^2(t_i^f)\right) e^{-\frac{t - t_i^f}{\tau_y^2}}$$

Post-synaptic weight update

$$\Delta w_{ij}^{+} = F_{+}(w_{ij}) x_j(t_i^f) y_i^2(t_i^{f-})$$



Pfister, J. P., & Gerstner, W. (2006). Triplets of spikes in a model of spike timing-dependent plasticity. The Journal of Neuroscience : The Official Journal of the Society for Neuroscience, 26(38), 9673–82.

SpiNNaker - Pair traces

timing_pair_impl.h
line 7

```
typedef int16_t post_trace_t;
```

timing_pair_impl.h
lines 46-49

```
static inline post_trace_t timing_get_initial_post_trace()
{
  return 0;
}
```


SpiNNaker - Triplet traces

timing_triplet_impl.h
line 7

```
typedef struct post_trace_t
{
  int16_t y1;
  int16_t y2;
} post_trace_t;
```

timing_triplet_impl.h
lines 46-49

```
static inline post_trace_t timing_get_initial_post_trace()
{
  return (post_trace_t){.y1 = 0, .y2 = 0};
}
```


SpiNNaker - Pair trace update

timing_pair_impl.h
lines 54-66

$$y_i(t) = 1 + y_i(t_i^f)e^{-\frac{t-t_i^f}{\tau_y}}$$

```
// Get time since last spike
uint32_t delta_time = time - last_time;

// Decay previous trace (y)
int32_t new_y = STDP_FIXED_MUL_16X16(last_trace,
  DECAY_TAU_Y(delta_time));

// Add energy caused by new spike to trace
new_y += STDP_FIXED_POINT_ONE;

log_debug("\tdelta_time=%d, y=%d\n", delta_time, new_y);

// Return new trace_value
return (post_trace_t)new_y;
```


SpiNNaker - Triplet trace update

timing_triplet_impl.h
lines 77-87

$$y_i^2(t) = \left(1 + y_i^2(t_i^f)\right)e^{-\frac{t-t_i^f}{\tau_y^2}}$$

```
// Y2 is sampled in timing_apply_post_spike BEFORE the spike
// Therefore, if this is the first spike, y2 must be zero
int32_t new_y2;
if(last_time == 0)
{
  new_y2 = 0;
}
// Otherwise, add energy of spike to last value and decay
else
{
  new_y2 = STDP_FIXED_MUL_16X16(
    last_trace.y2 + STDP_FIXED_POINT_ONE,
    DECAY_TAU_Y2(delta_time));
}
```

SpiNNaker - Pair weight update

timing_pair_impl.h
lines 136-150

$$\Delta w_{ij}^{+} = F_{+}(w_{ij}) x_j(t_i^f)$$

```
uint32_t delta_t = time - last_pre_time;

// If spikes are not co-incident
if (delta_t > 0)
{
  // Calculate x(time) = x(last_pre_time) * e^(-delta_t/tau_x)
  int32_t x = STDP_FIXED_MUL_16X16(last_pre_trace,
    DECAY_TAU_X(delta_t));

  log_debug("\t\t\tdelta_t=%u, x=%d\n",
    delta_t, x);

  // Apply potentiation to synapse state
  return weight_one_term_apply_potentiation(previous_state, x);
}
```

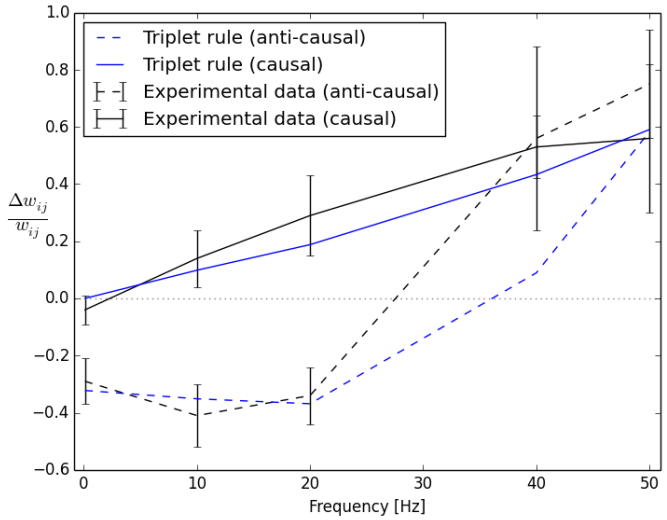

SpiNNaker - Triplet weight update

timing_triplet_impl.h
lines 165-179

$$\Delta w_{ij}^{+} = F_{+}(w_{ij}) x_j(t_i^f) y_i^2(t_i^{f-})$$

```
if (delta_t > 0)
{
  // Calculate x(time) = x(last_pre_time) * e^(-delta_t/tau_x)
  int32_t x = STDP_FIXED_MUL_16X16(last_pre_trace,
    DECAY_TAU_X(delta_t));

  // Multiply this by y2(time) to get triplet term
  int32_t x_y2 = STDP_FIXED_MUL_16X16(x, trace.y2);

  log_debug("\t\t\tdelta_t=%u, x=%d, y2=%d, x_y2=%d\n",
    delta_t, x, trace.y2, x_y2);

  // Apply potentiation to synapse state
  return weight_one_term_apply_potentiation(previous_state, x_y2);
}
```


Triplet model

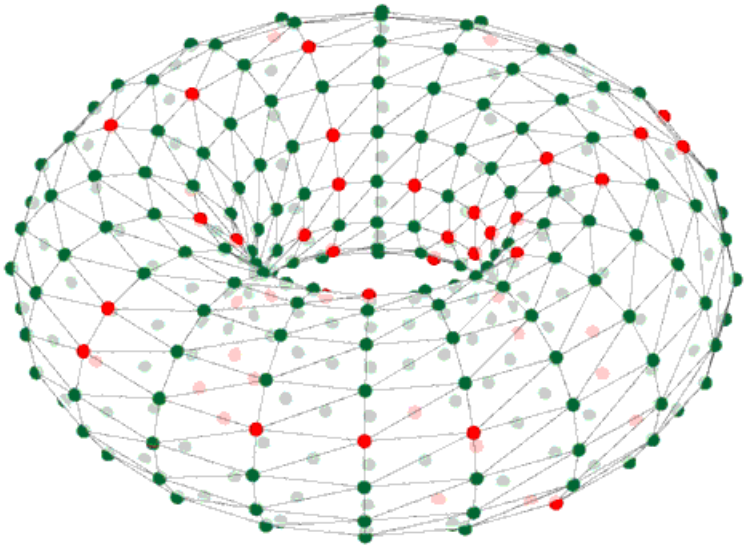


Thank you!

Any questions?

james.knight@manchester.ac.uk

External Devices



Alex Rast, Sergio Davies, Alan Stokes

SpiNNaker Workshop
September 2015

European Research Council
Established by the European Commission

Human Brain Project

EPSRC

SpiNNaker

MANCHESTER 1824

Outline

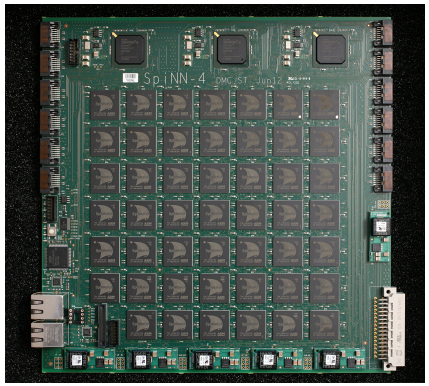
Ethernet Protocol
- Protocol Format
- The EIEIO Implementation

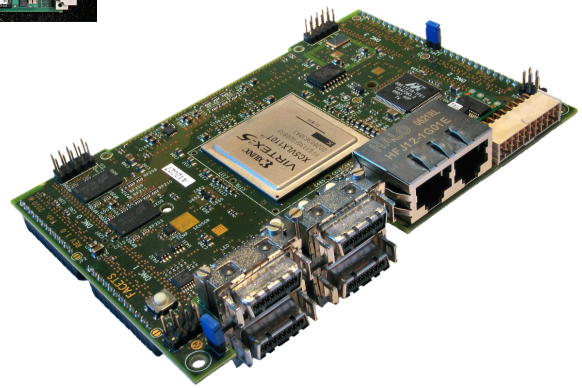
External Devices - Ethernet
- Live Output
- Live Input
- Prefixes and Keys
- The Database Interface

Worked Example: spike_io.py
- Setup
- Populations and Projections
- Setting up a Host Device
- Getting Input/Output

External Devices - SpiNNaker Link
- Hardware Connection
- Toolchain Modifications
- PyNN Instantiation

Connecting to External Devices





1: Spike Injection From a Synthetic Source Useful for very large spike lists that won't fit on-chip.

2: Live Spike Output for Visualisation, Etc. Useful for interactive display of results.

3: Spikes From an External AER Device Useful for Hardware With UDP Support

4: Spikes To an External AER Device Useful for SpiNNaker-based preprocessing

5: Bidirectional Communication Useful for ad-hoc real-time interactive systems

Fig. 1: Real-Time Systems

Available Interfaces

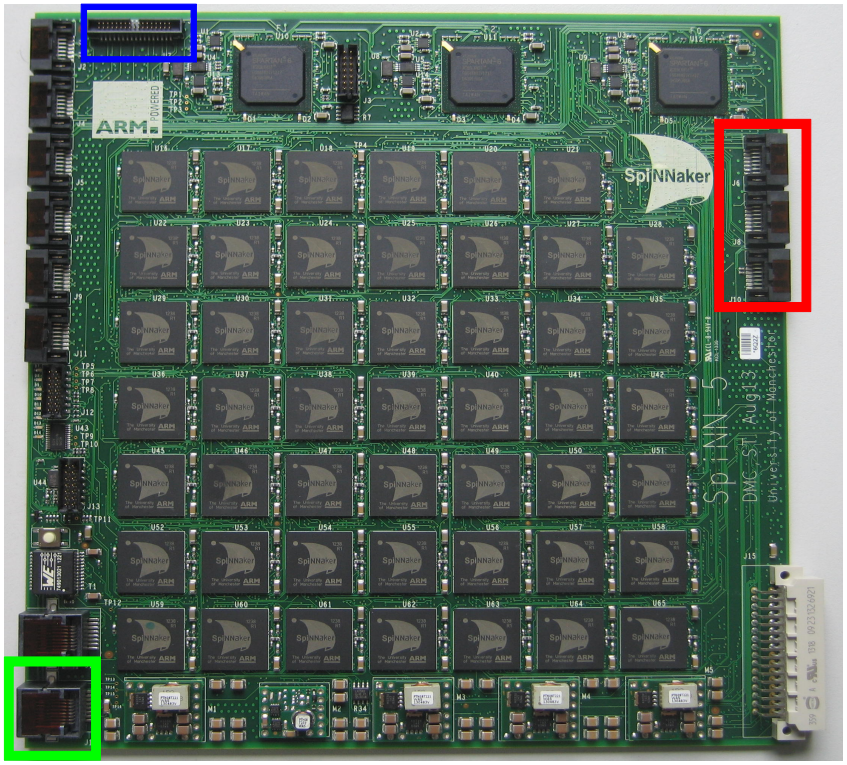


Fig. 2: SpiNNaker Interfaces

1: Ethernet Relatively slow-speed connections using a UDP protocol. Relatively "plug-and-play".

2: SpiNNaker Link Direct native connection to the SpiNNaker fabric. Usually uses an intervening FPGA. Some support available in tools but expect some low-level work.

3: "SpiNNLink" Not a SATA-compliant interface but uses SATA connectors and cabling. High-speed. "Roll-your-own"!

Ethernet AER Protocol

Packet Features

Header	Command Information
15 0	...

Header	Key	...	Key
15 0	31/15 0	...	31/15 0

Header	Key	Payload	...	Key	Payload
15 0	31/15 0	31/15 0	...	31/15 0	31/15 0

Header	Prefix	Key	...	Key
15 0	15 0	31/15 0	...	31/15 0

Header	Prefix	Key	Payload	...	Key	Payload
15 0	15 0	31/15 0	31/15 0	...	31/15 0	31/15 0

Header	Fixed Payload	Key	...	Key
15 0	31/15 0	31/15 0	...	31/15 0

Header	Payload Base	Key	Payload	...	Key	Payload
15 0	31/15 0	31/15 0	31/15 0	...	31/15 0	31/15 0

Header	Prefix	Fixed Payload	Key	...	Key
15 0	15 0	31/15 0	31/15 0	...	31/15 0

Header	Prefix	Payload Base	Key	Payload	...	Key	Payload
15 0	15 0	31/15 0	31/15 0	31/15 0	...	31/15 0	31/15 0

Fig. 3: AER Packet Formats

Multiple Spikes/Packet:
Up to 256 Spikes (64 on SpiNNaker)

Optional Payloads:
Each spike may include a second word as a payload

Embeddable Commands:
Device-specific commands can be put in the stream

Selectable Word Width:
Addresses and payloads can be 16- or 32-bit

Fire and Forget:
No handshaking support.

AER Protocol Headers

0	0	D	T	W	Y	Tag		Count		Data	
15	14	13	12	11	10	9	8	7	0	15	0

P	0	D	T	W	Y	Tag		Count		Key Prefix (if P) or Payload Prefix (if D)	
15	14	13	12	11	10	9	8	7	0	15	0

1	F	1	T	W	Y	Tag		Count		Key Prefix		Payload Prefix	
15	14	13	12	11	10	9	8	7	0	15	0	31/15	0

Fig. 4: AER Data Packet Headers

0	1	Command		(Device-specific)
15	14	13	0	...

Fig. 5: AER Command Packet Header

Spike Count:
Bits 7-0 of the header

Word Width:
Bit 11 (32-bit = 1)

Data Payloads:
Bit 10 (1 = Enable)

Command Packets:
Bits 15-14 (Command = 01)

Key Prefixes:
ORs a fixed key offset. Bit 15 enables, Bit 14 sets which halfword to OR with. Prefix follows header.

Payload Prefixes:
Enables a fixed payload OR pattern. Bit 13 enables. Prefix follows any key prefix.

Timestamps:
Bit 12 (1 = Payloads are timestamps)

EIEIO: An AER Compliant Implementation

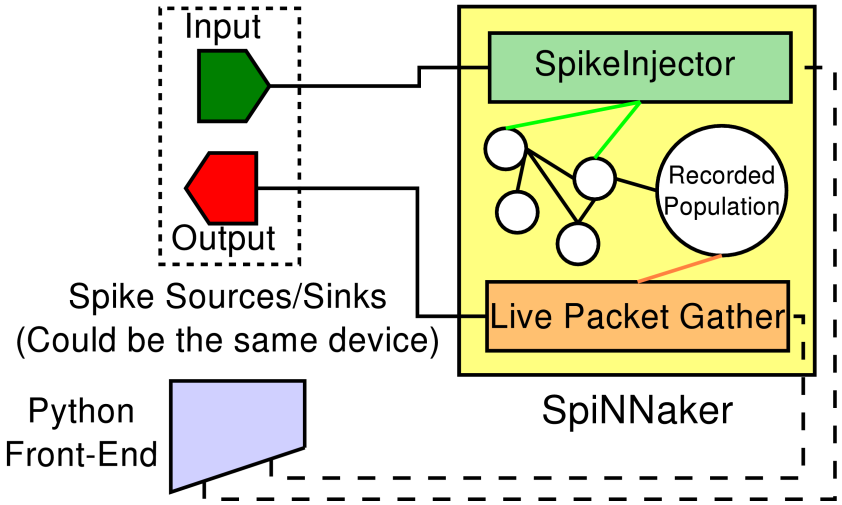


Fig. 3: EIEIO Architecture

2 Main Modules Plus a Python Front-End

SpikeInjector:
UDP-over-Ethernet AER interface for input from external devices

LivePacketGatherer:
UDP-over-Ethernet spike capture and external retransmission

Python Front End:
Two methods:
activate_live_output_for(pop)
sets a LivePacketGatherer.
SpikeInjector(n_neurons, **args)
instantiates a
ReverseIPTagMulticastSource

Live output functionality

Example PyNN script:

```
import pyNN.spiNNaker as p
p.setup (timestep = 1.0)
spike_times = {'spike_times': [[0]]}
spike_source = p.Population(
                1, p.SpikeSourceArray, spike_times, label="ss1")
import spynnaker_external_devices_plugin.pyNN as ext_dev
ext_dev.activate_live_output_for(spike_source)
p.run(100)
```

Default output: 32-bit EIEIO packets with timestamp prefix to identify the time of the event(s)

User-configurable

Configuration options

Live output functionality can be configured:

Parameter	Default	Description
`population`	`-`	PyNN population that will send spikes to the output world
`host, port`	`From .cfg`	Target host IP and UDP port for the packets sent
`use_prefix`	`False`	Use a key prefix in packet header
`key_prefix`	`None`	Key prefix to use
`prefix_type`	`None`	Upper / Lower halfword for key prefix
`message_type`	`32 bit`	16 / 32 bit keys and payloads
`right_shift`	`0`	If keys to be sent are 16 bits, 32-bit keys from SpiNNaker may be right-shifted
`payload_as_time_stamps`	`True`	Payload indicate time stamp of events
`use_payload_prefix`	`True`	Use a payload prefix in packet header
`payload_prefix`	`None`	Payload prefix to use

Processing received spikes

SpynnakerLiveSpikesConnection provides a Python host-based virtual device

Example PyNN script:

```
import pyNN.spiNNaker as p
from spynnaker_external_devices_plugin.pyNN.connections.\
                        spynnaker_live_spikes_connection import
                        SpynnakerLiveSpikesConnection

def processing(label, time, neuron_ids):
    print label, time, neuron_ids

p.setup (timestep = 1.0)
spike_source = p.Population(
       1, p.SpikeSourceArray, {'spike_times': [[0]]}, label='ss1')
import spynnaker_external_devices_plugin.pyNN as ext_dev
ext_dev.activate_live_output_for(spike_source)
live_spikes_connection = SpynnakerLiveSpikesConnection(
                                        receive_labels=['ss1'])
live_spikes_connection.add_receive_callback('ss1', processing)
p.run(100)
```

Configuring live connections

SpynnakerLiveSpikesConnection allows a few parameters for configuration:

Parameter	Default	Description
`receive_labels`	`None`	Label(s) of the population(s) from which spikes are received
`send_labels`	`None`	Label(s) of the population(s) to which spikes will be sent
`local_port`	`19999`	(optional) UDP port on which to listen for incoming messages
`local_host`	`None`	IP address of the interface to listen for incoming messages

Live spike injection

SpikeInjector is a piece of simulation software which runs on SpiNNaker and receives data packet from the external world

Example PyNN script:

```
import pyNN.spiNNaker as p
import spynnaker_external_devices_plugin.pyNN as ext_dev
p.setup (timestep = 1.0)
spike_source = p.Population(
          1, ext_dev.SpikeInjector, {'port':17899}, label='ls1')

ext_dev.activate_live_output_for(spike_source)
p.run(100)
```

In this example, SpiNNaker listens on UDP port 17899 for EIEIO packets to parse.

Synchronising the injector

The injector may be synchronised with the start of the simulation on SpiNNaker using a callback set up for the purpose. Example:

```
def start_sender(label, sender):
    print 'start sending', label, sender

import pyNN.spiNNaker as p
import spynnaker_external_devices_plugin.pyNN as ext_dev
from spynnaker_external_devices_plugin.pyNN.connections.\
                        spynnaker_live_spikes_connection import
                        SpynnakerLiveSpikesConnection

p.setup (timestep = 1.0)
spike_source = p.Population(
           1, ext_dev.SpikeInjector, {'port':17899}, label='ls1')
ext_dev.activate_live_output_for(spike_source)
live_spikes_connection = SpynnakerLiveSpikesConnection(
                                        send_labels=['ls1'])
live_spikes_connection.add_start_callback('ls1', start_sender)
p.run(100)
```

Operations on live connections

Function	Description
`add_receive_callback`	Add a function call upon receiving a message from SpiNNaker
`add_start_callback`	Add a function call upon start of the simulation
`send_spike`	Send an EIEIO packet with a single key to SpiNNaker
`send_spikes`	Send an EIEIO packet with multiple keys to SpiNNaker

Neuron ID vs Routing key

- In a population, neurons are numbered from 0
- In the SpiNNaker machine neurons are globally identified by their routing key (32 bit)
- Remaining bits identify the source population ID between populations

Translation

How to perform the translation?

- Send only the neuron ID to the population (conversion performed on the board)
  (sending spikes to SpiNNaker)

- Use “send_spike” and “send_spikes” functions (conversion performed on the host)
  (sending spikes to SpiNNaker)

- Interface to the database (conversion performed on the host)
  (sending AND receiving spikes to/from SpiNNaker)

Neuron ID transmission

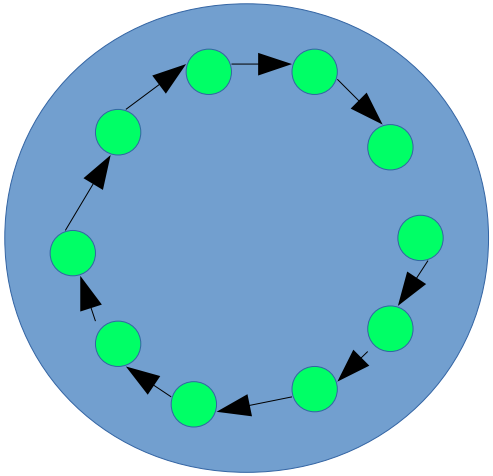
- Key 32 bit: No translation performed – assumed to be fully specified key
- Key 16 bit, key prefix: No translation performed – the combination of prefix and key specifies the key
- Key 16 bit, no key prefix: Translation performed to the virtual key space assigned to the injector population

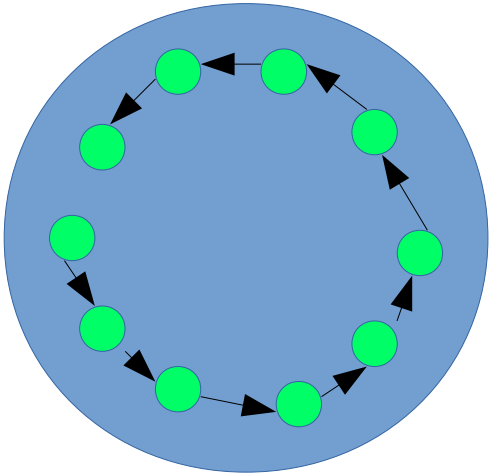
Example – spike_io.py – 1

```
import spynnaker.pyNN as Frontend
import spynnaker_external_devices_plugin.pyNN as ExternalDevices
from spynnaker_external_devices_plugin.pyNN.connections\
    .spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection

Frontend.setup(timestep=1.0, min_delay=1.0, max_delay=144.0)

n_neurons = 100

cell_params_lif = {'cm': 0.25,
                   'i_offset': 0.0,
                   'tau_m': 20.0,
                   'tau_refrac': 2.0,
                   'tau_syn_E': 5.0,
                   'tau_syn_I': 5.0,
                   'v_reset': -70.0,
                   'v_rest': -65.0,
                   'v_thresh': -50.0}

cell_params_spike_injector = {'port': 12345}

cell_params_spike_injector_with_key = {
    'port': 12346,
    'virtual_key': 0x70000}
```

Example – spike_io.py – 2

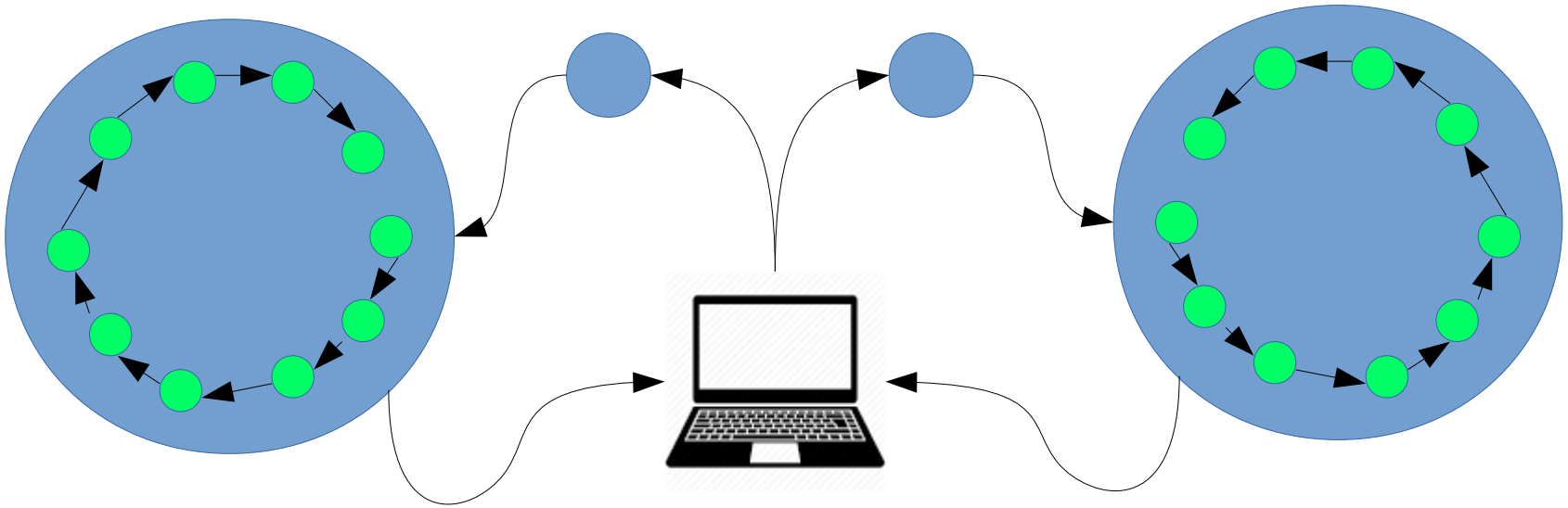
```
pop_forward = Frontend.Population(n_neurons, Frontend.IF_curr_exp,
                                  cell_params_lif, label='pop_forward')
pop_backward = Frontend.Population(n_neurons, Frontend.IF_curr_exp,
                                   cell_params_lif, label='pop_backward')

loop_forward = list()
loop_backward = list()
for i in range(0, n_neurons - 1):
    loop_forward.append((i, (i + 1) % n_neurons, weight_to_spike, 3))
    loop_backward.append(((i + 1) % n_neurons, i, weight_to_spike, 3))
Frontend.Projection(pop_forward, pop_forward,
                    Frontend.FromListConnector(loop_forward))
Frontend.Projection(pop_backward, pop_backward,
                    Frontend.FromListConnector(loop_backward))
```





Example – spike_io.py – 3

```
injector_forward = Frontend.Population(n_neurons, ExternalDevices.SpikeInjector,
    cell_params_spike_injector_with_key, label='spike_injector_forward')
injector_backward = Frontend.Population(n_neurons, ExternalDevices.SpikeInjector,
    cell_params_spike_injector, label='spike_injector_backward')

Frontend.Projection(injector_forward, pop_forward,
                    Frontend.OneToOneConnector(weights=weight_to_spike))
Frontend.Projection(injector_backward, pop_backward,
                    Frontend.OneToOneConnector(weights=weight_to_spike))

ExternalDevices.activate_live_output_for(pop_forward,
database_notify_host="localhost", database_notify_port_num=19996)
ExternalDevices.activate_live_output_for(pop_backward,
database_notify_host="localhost", database_notify_port_num=19996)
```



Example – spike_io.py – 4

```
live_spikes_connection_sending = SpynnakerLiveSpikesConnection(
    receive_labels=None, local_port=19999,
    send_labels=["spike_injector_forward", "spike_injector_backward"])

live_spikes_connection_sending.add_start_callback("spike_injector_forward",
                                                  send_input_forward)
live_spikes_connection_sending.add_start_callback("spike_injector_backward",
                                                  send_input_backward)



live_spikes_connection_receive = SpynnakerLiveSpikesConnection(
    receive_labels=["pop_forward", "pop_backward"],
    local_port=19996, send_labels=None)

# Set up callbacks to occur when spikes are received
live_spikes_connection_receive.add_receive_callback("pop_forward",
                                                    receive_spikes)
live_spikes_connection_receive.add_receive_callback("pop_backward",
                                                    receive_spikes)
```

Example – spike_io.py – 5

```
# Create a sender of packets for the forward population
def send_input_forward(label, sender):
    for neuron_id in range(0, 100, 20):
        time.sleep(random.random() + 0.5)
        print_condition.acquire()
        print "Sending forward spike", neuron_id
        print_condition.release()
        sender.send_spike(label, neuron_id, send_full_keys=True)


# Create a sender of packets for the backward population
def send_input_backward(label, sender):
    for neuron_id in range(0, 100, 20):
        real_id = 100 - neuron_id - 1
        time.sleep(random.random() + 0.5)
        print_condition.acquire()
        print "Sending backward spike", real_id
        print_condition.release()
        sender.send_spike(label, real_id)


# Create a receiver of live spikes
def receive_spikes(label, time, neuron_ids):
    for neuron_id in neuron_ids:
        print_condition.acquire()
        print "Received spike at time", time, "from", label, "-", neuron_id
        print_condition.release()
```

Example – spike_io.py – 6

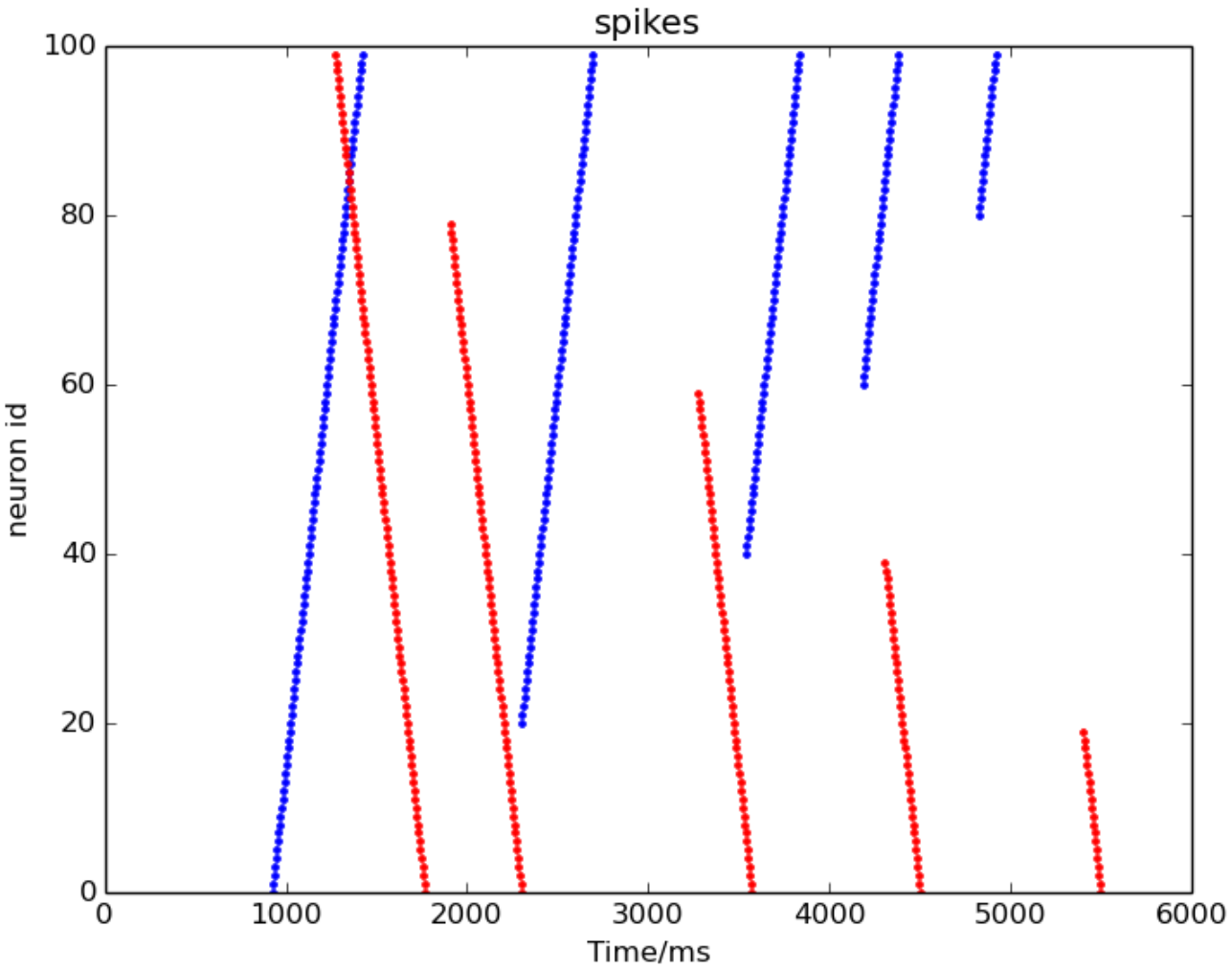


Fig4: Output of the spike_io.py network

External Devices Through The SpiNNaker Link

- A method of direct connection using native AER links
- Usually requires an FPGA interface board
- Supported in the toolchain via "virtual chips"

Fig4: FPGA connector

How to connect devices to a SpiNNaker board

- Connect the device to the SpiNNaker link connector

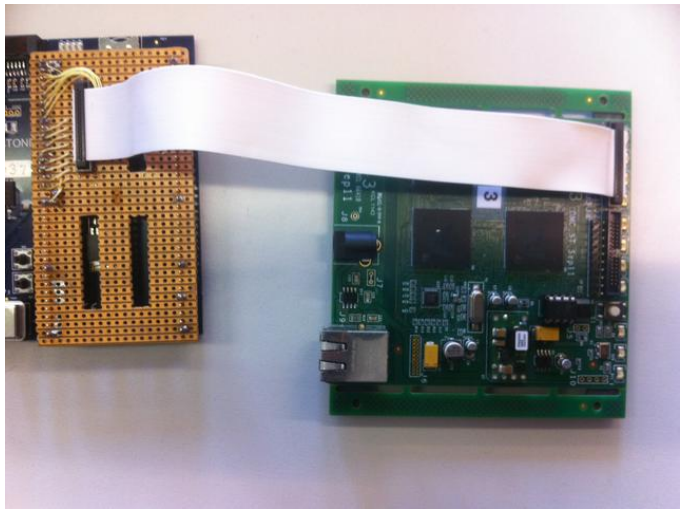


Fig5: connecting to a spinn-3 Board

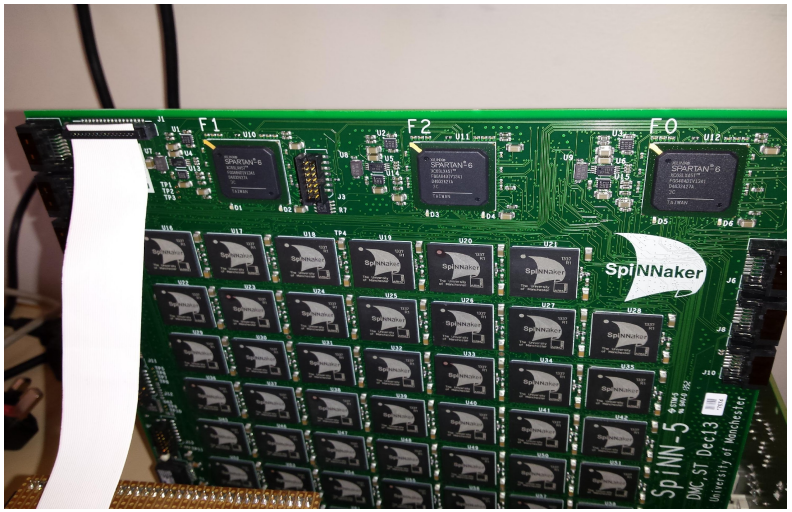


Fig6: connecting to a spinn-5 Board

How external devices are represented in sPyNNaker

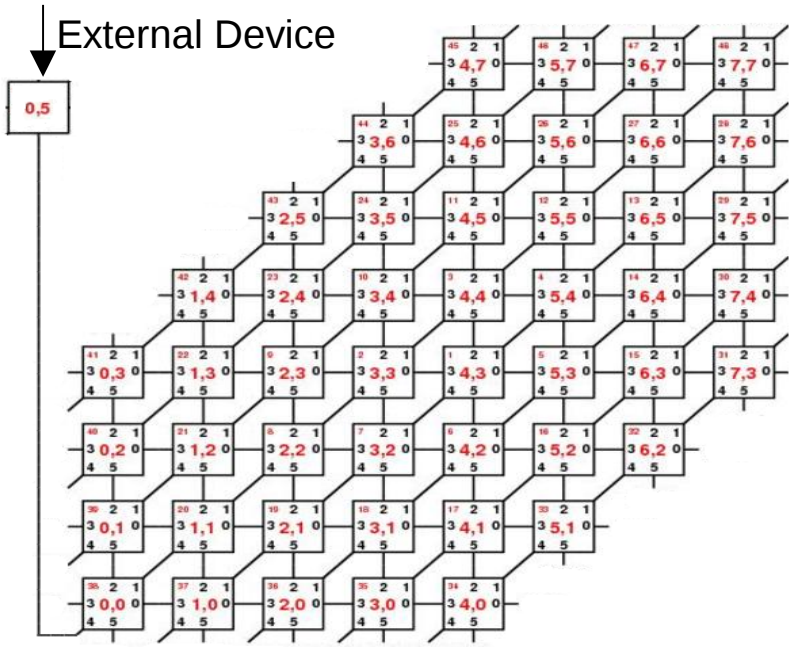


Fig7: Structure for either a Spinn-4 or Spinn-5 boards

1. External devices are represented in sPyNNaker by virtual ID's.

2. Each virtual ID must occupy a chip address space not currently used by the machine ([0,5],[0,6],[0,7],[5,2], etc.)

3. Each virtual chip is routed to a real chip (chip [0,0] in this case)

4. The connection to the real chip must be through a link not currently used by the real chip (links [W,SW,S] in this case)

External devices: Pacman

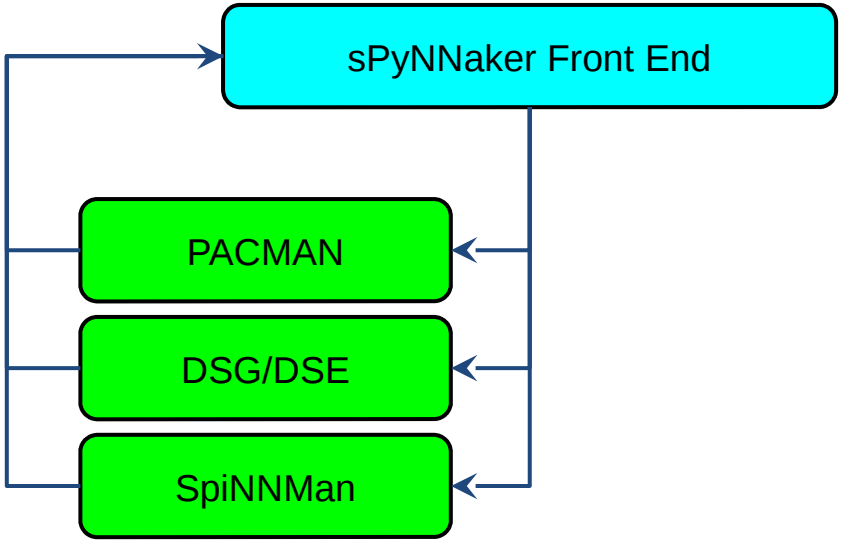


Fig8: The sPyNNaker stack

2.1 Partitioning would produce one subvertex *SV* with all the neurons

2.2 Placement would place *SV* into the virtual address range defined for the external device.

2.3 Routing will treat the virtual addresses as if they were normal ones and route stuff accordingly.

External devices: DSG/SpinnMan

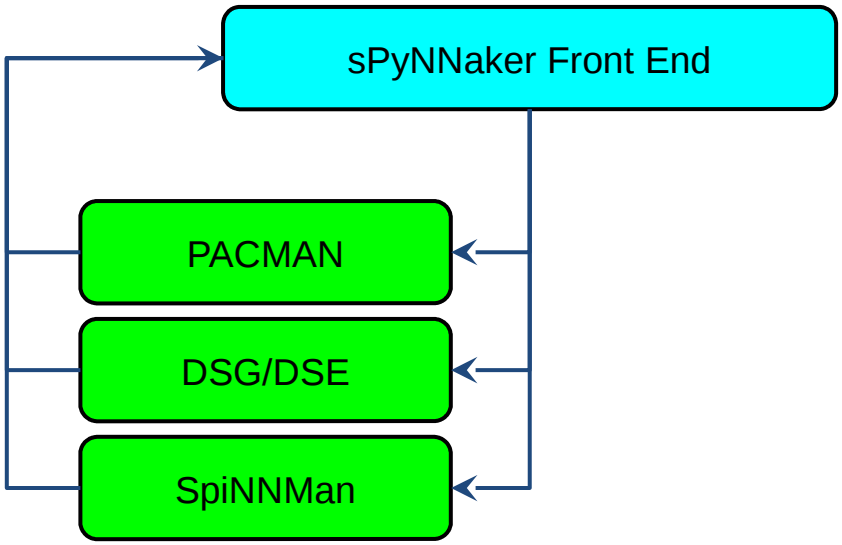


Fig8 (*redux*): The sPyNNaker stack

3.1 No data spec is generated for the virtual device

3.2.1 No data spec is therefore executed for models running on the external device

3.2.2 Data specs are generated for models running on real spinnaker cores

3.3 Should occur correctly.

SpinnMan does not change

Adding a new external device

Calls from PYNN

```
import pynn.spinnaker as p
p.setup( timestep=1.0,
         min_delay = 1.0,
         max_delay = 32.0)
external_device_requirements = {
   ‘spinnaker_link’:0,
     'XXXXXXX'=XXXXXX}
external_device =
   p.Population(1, p.New_External_device,
                    external_device_requirements,
                    label='External retina'))
external_device.record()
p.run(10000)
```

An example script

Differences

Step3:
Add any new parameters that the new device model requires

Step4:
Initializes the new device as a population

Coming Soon...

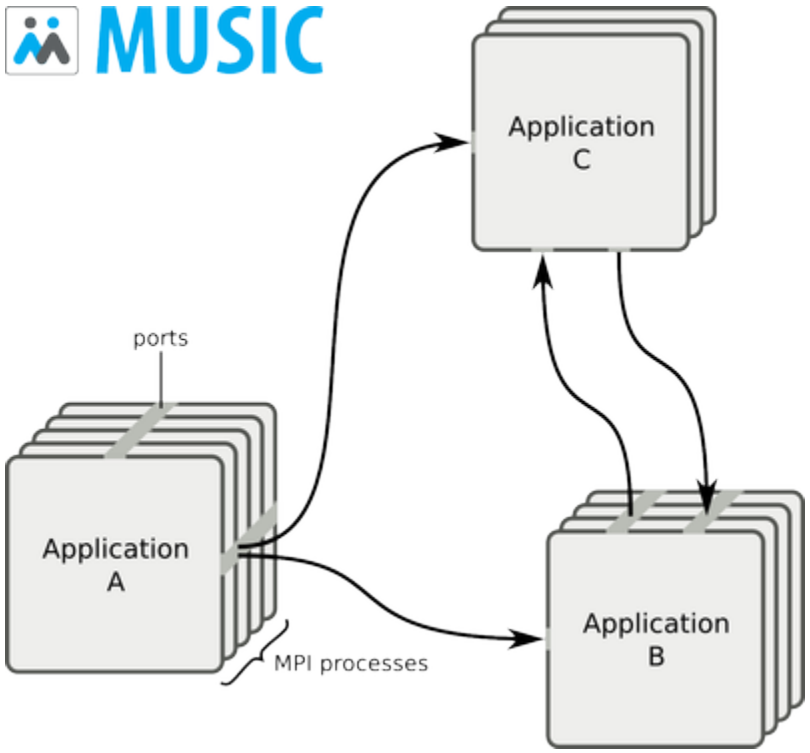


Fig. 9 MUSIC
(Image from Mikael Djurfelt and the MUSIC Project 2015)

Updated AER Protocol:
Will support keys, payloads, and timestamps in the same packet, 64-bit data types.

MUSIC interface:
A universal protocol for intersystem communications

5th SpiNNaker Workshop

Day 5

September 11th 2015

Time	Session	Owner
09:00	Free lab time	
10:30	Coffee	
11:00	Free lab time	
12:00	Lunch and close	

Manchester, UK